AFRL-IF-WP-TR-2000-1506

THE STANFORD IBM MANAGEMENT OF MULTIPLE INFORMATION SYSTEMS (TSIMMIS) – IBM ALMADEN RESEARCH CENTER

ALLEN W. LUNIEWSKI, PH.D.
LAURA HAAS, PH.D.

IBM ALMADEN RESEARCH CENTER
650 HARRY ROAD
SAN JOSE, CA 95120-6099

MARCH 2000

FINAL REPORT FOR 09/30/1993 – 09/30/1999

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE OH 45433-7334

NOTICE

Using Government drawings, specifications, or other data included in this document are for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell patented invention that may relate to them.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Charles P. Satterthwaite

CHARLES P. SATTERTHWAITE, Project Engineer
Embedded Information System Engineering Branch
AFRL/IFTA

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Information System Engineering Branch
AFRL/IFTA

Eugene C. Blackburn

EUGENE C. BLACKBURN, Chief
Information Technology Division
AFRL/IFT

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
MARCH 2000

3. REPORT TYPE AND DATES COVERED
FINAL REPORT FOR 09/30/1993 - 09/30/1999

4. TITLE AND SUBTITLE
THE STANFORD IBM MANAGEMENT OF MULTIPLE INFORMATION SYSTEMS (TSIMMIS) -- IBM ALMADEN RESEARCH CENTER

5. FUNDING NUMBERS
C F33615-93-C-1337
PE 62301
PR A523
TA 01
WU 01

6. AUTHOR(S)
ALLEN W. LUNIEWSKI, PH.D.
LAURA HAAS, PH.D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IBM ALMADEN RESEARCH CENTER
650 HARRY ROAD
SAN JOSE, CA 95120-6099

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7334
POC: CHARLES P. SATTERTHWAITE, AFRL/IFTA, 937-255-6548 EXT. 3584

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-WP-TR-2000-1506

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

OBJECTIVE to improve the access and utility of heterogeneously distributed information.
BACKGROUND DARPA awarded this effort as sister effort to TSIMMIS-STANFORD. This work develops techniques to access, integrate, and utilize distributed heterogeneous information.

14. SUBJECT TERMS
Heterogeneous distributed information, Intelligent Query Processing, Databases

15. NUMBER OF PAGES
111

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
SAR

TABLE OF CONTENTS

Section 1: SUMMARY of RESEARCH

This report is a compilation of a summary of the research work and publications produced as a result of the project: "An Integrated Information Management System", ARPA Contract # F33615-93-C-1337, which was conducted at IBM's Almaden Research Center from 1994-1999 with Allen Luniewski as Principal Investigator and Laura Haas as co-principal investigator. Other participants in this project were regular IBM employees (Peter Schwarz, Mary Tork-Roth, Bartholomew Niswonger), a number of students working temporarily at IBM (Neal Palmer, Paul Aoki, Fatma Ozcan, Ioana Ursu), and post-docs (Markus Tresch, Andreas Geppert). This work was conducted as part of DARPA's I3 (Intelligent Integration of Information) program.

This project attacked two problems of relevance to the I3 program: the classification of semi-structured data to allow ingestion of that data into a data management system, and the creation of "wrapper" technology to allow integration of multiple, disparate, information sources into a central query engine.

The classification work was intended to develop technologies to allow the automatic classification of semi-structured data into a given type (or class) taxonomy. This work was motivated by the observation made in the Rufus project [VLDB-93] that the vast majority of digital information was not contained in databases. To allow effective ingestion of that data into a data management system to allow effective mechanisms for organizing, searching and operating upon that data, it is necessary to determine the type of the data. The nature of semi-structured data means that the type of that information is not inherently known and must, instead, be determined. The general approach taken to automatically determine the type of semi-structured data was to define a set of *features* that might characterize semi-structured information (e.g., file name, presence of keywords in text) and use these features to define a feature space in which all documents lie. Within that space, points are defined for each type (called "centroids"). The type of the data is then determined to be the type represented by the centroid that is the minimum of a distance metric from the data to all of the centroids. A number of areas were investigated as part of this research. One area of research was to develop algorithms and heuristics to determine a good set of features. A second area was to determine which distance metric works best in this application. We also developed a metric that allows an estimation of how good the results of the classification are. Finally, we explored techniques for taking a classifier and extending it with additional types of data. This work is discussed in more detail in Section 2 and is described in detail in reprints of the conference papers that describe this work that are included in this report.

The Garlic (and TSIMMIS) project(s) focused on a different approach to making non-database data accessible via high-level queries. Rather than try to capture more data in databases as in Rufus, these projects introduced middleware systems that allow access to the data using the native capabilities of its existing storage mechanism via a central query engine. Wrappers are the portion of the system which encapsulates a data source. A wrapper's role is two-fold: to describe the data in its data source and the query capabilities of the wrapper/data source pair, so that the query engine knows what kinds of queries it may ask, and to translate queries from the engine into actions the underlying data source can perform to return the desired data. We examined a number of areas of wrapper architecture as part of this research. A key concern was the interface between the wrappers and the middleware query engine, especially how information about query capabilities is passed from wrapper to middleware. We developed a novel approach that allows the wrapper to actively participate in query planning. Once the wrapper can participate in query planning, it is natural to consider asking it to provide information on the cost of a possible plan. The ability to get this information from the wrapper makes the system much more extensible, yet puts a major burden on the wrapper creator to understand and model the source system. Thus a second major area of research was on an infrastructure for supporting wrapper cost estimates. In addition to these major thrusts, we prototyped a number of wrappers, including a flexible wrapper for web data sources. Our experiences led us to further investigate additional support for wrappers. In particular, we explored the use of a cache manager in the middleware to relieve the wrapper of the burden of caching expensive-to-access data. We also defined a simple language for declaring equivalent path expressions, information which allows the optimizer to choose a better overall execution plan for a query. We cover this work in more detail in Section 3, and in the reprints of conference papers that accompany this report.

Section 2: FILE CLASSIFICATION

The file classification problem can be summarized as: given a piece of (semi-structured) information, and given a set of types, determine which type is the best match for that data. Much of the focus of our work was on semi=strctured text documents but it applies to non-textual data as well. The general approach that we took is based upon the vector space model of information retrieval.

The vector space model consists of a feature space, a colleciton of centroids for the type taxonomy and a distance metric. The feature space is an N-dimensional space defined by a colleciton of N features. The features are determined by a human analyst. For each member of the type taxonomy a centroid in this space is determined by analyzing a set of pre classified training data. Finally, a distance metric is chosen. Classification proceeds as follows. When a document is presented for classification, the value of the N features for that document are determined. This defines a point in the feature space. The distance metric is then applied to determine the distance from this point to each centroid. The centroid closest, under the distance metric, to the data point determines the type of the data. This is discussed in more detail in [VLDB-95].

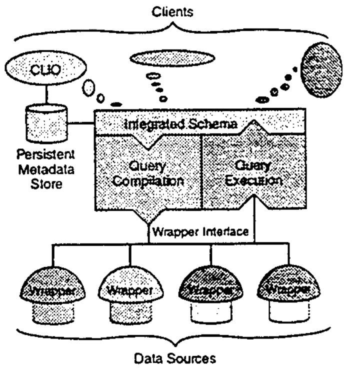
In [VLDB-95], this process was evaluated in the context of the data classification process. Key results are as follows. First, the cosine metric as used in traditional information retrieval systems performs best in this application also. Other metrics were found to either be less accurate or less stable as the training set and test data changed. Second, we discovered the existence of a *confidence metric* that gives an indication of how good a job the classifier did in determining the type of some data. If the confidence metric is sufficiently low, there is good reason to doubt the result of the classification and a human expert may need to be consulted. Third, we explored the training problem for the classifier. We determined that relatively small numbers of training documents are required to produce an effective classifier. Moreover, the confidence metric can be used to determine which documents are best added to the training set. Fourth, we explored feature selection. A large number of potential features exist for a given type taxonomy. We discovered that limiting the number of features resulted in a more effective classifier than with the larger number of features. We defined the *discriminating power* of a feature which was the basis for techniques for determining which features to include, as well as techniques for combining features.

In [CIKM-95] we investigated the problem of extending a classifier. Consider a classifier that distinquishes between M types. Suppose that after this classifier is built, it is desired to add additional types to the set of types distinguished. If the initial training data is available, this extension is easy - the classifier construction algorithms can be directly executed. Suppose, however, that the training data is not available (e.g., the classifier is a product and the training data is considered proprietary). In this paper we explored techniques for building extensible classifiers and for doing the extension. We also explored an intermediate position in which abstracts of training data were made available. Highly effective classifiers could be constructed from these abstracts while limiting the information about the initial training set that had to be released.

These various research activities allow the creation of effective classifiers. The algorithms we explored support tools to help a human analyst create these classifiers with some rigor. Classifiers such as those that we explored in this work are inherently imprecise. We feel that this work represents a significant contribution to the quality of these classifiers.

Section 3: WRAPPERS

The figure below shows a typical database middleware system architecture, as exemplified by Garlic [RIDE-DOM'95]. Garlic is a query engine that optimizes and executes queries over diverse data sources posed in an object-extended SQL. Garlic is capable of executing any extended SQL operation against data from any source. In both planning and executing the query, it communicates with wrappers for the various data sources involved in the query.



A wrapper hides the details of the data source's interface and enables access to the data source using the middleware's internal protocols. Our research activities have been directed at designing a wrapper architecture that is flexible (able to wrap any data source), productive (wrappers can be written quickly and maintained easily), and effective (data can be retrieved efficiently and the search and data manipulation capabilities of the source utilized), and which supports the optimization of queries over data in multiple sources.

Our wrapper architecture includes four main interfaces, as described in [VLDB-97]. The first interface allows the data in the source to be described to the middleware. Data are described using Garlic's data model, an object-oriented model based on the ODMG standard [ODMG]. Data in the source are viewed as objects, and Garlic refers to these objects using an OID it manufactures based on the source, the object's type, and a unique key determined by the wrapper. This OID allows the middleware to apply methods on objects; from the OID, the middleware can determine the appropriate wrapper, and the wrapper can locate the necessary data and apply the method. The wrapper also defines object collections (the targets of queries in Garlic). Using the second interface, wrappers provide methods to get the value of each attribute of an object, and to encapuslate any specialized search capabilities of the source. (These methods are typically implemented as commands in the native language or programming interface of the underlying source). The third interface provided by the wrapper allows the middleware query engine to open an iterator over a collection or query result.

The fourth interface provides a description of the source's query processing capabilities. This description consists of a set of planning methods [VLDB-97]. Different sources may vary greatly in their query processing capabilities, and thus will provide different planning methods. For example, a wrapper for relational databases can typically handle multi-way joins as well as selections and projections on a single relation, hence the relational wrapper will provide methods for planning both single-collection accesses and joins. On the other hand, a wrapper for an image processing system that only handles requests to a single collection of images would provide only a method for planning single-collection accesses. A wrapper does not have to reflect the full query functionality of its data sources. However, in order for the data in that data source to be accessible through queries, some minimum functionality must be provided, i.e., at least one planning method.

This fourth, query processing interface is a unique contribution of our research. The interface allows a query optimizer to communicate with the wrapper during query optimization. The optimizer can describe a query that it would like the wrapper to handle. The wrapper can then indicate which part of the query it *will* handle, by returning one or more "plans". For example, the optimizer might ask the wrapper to access a particular collection and apply two predicates. If the wrapper can only handle a single predicate on that collection, it returns a list of plans to the optimizer, each of which applies only one of the predicates.

We experimented with and tested the wrapper architecture, by building a number of wrappers. In particular, we designed and implemented a relational database wrapper, which can be used against DB2 or Oracle databases, a Lotus Notes wrapper, which we applied to two different Notes databases, and a molecular database wrapper for the Daylight chemical structure search engine. In addition, we built a versatile web wrapper. This wrapper provides much of the basic functionality needed to wrap web data sources, and can be easily tailored to wrap a particular source. We tested the generality and power of this wrapper by applying it to several different web sites, including a movie database, a CD-ROM database, a hotel guide, and the BigBook yellow page site. To customize the wrapper for a particular site, the wrapper writer writes a grammar describing the HTML pages generated as answers by the site. Given this grammatical description, the wrapper can generate the necessary code for answering queries, etc. This experiment demonstrated the flexibility of the wrapper architecture, and the power of the layered approach to wrapper writing. Overall, this diversity of data sources provided a good test of the architecture and interfaces, and the results were very encouraging. Typically, data from a new source could be accessed through the wrapper within three days of the start of wrapper development. A finely tuned wrapper took at most a few weeks.

We also studied how the Garlic wrapper architecture could be applied to a commercial middleware product. The goal of this work was two-fold: to understand whether the wrapper architecture was complete and robust enough for commercial use, and as a first step to get the technology out to the world for its use. We worked closely with IBM's DataJoiner product team to compare their "data access modules" with our wrappers. Data access modules serve a similar role to wrappers, but in DataJoiner they must be created by DataJoiner developers, and adding a data access module requires that changes be made throughout the DataJoiner code base. This limits the flexibility and extensibility of the DataJoiner product. By replacing data access modules with wrappers, DataJoiner is able to dynamically add data sources in response to customer demand, and third parties will be able to create wrappers for other sources, increasing the "reach" of the DataJoiner middleware. Working with the product team, we designed data definition language (DDL) commands for creating new wrappers, and new catalogs to support the more flexible wrapper architecture. We also designed and implemented a version of the necessary interfaces between the wrappers and DataJoiner's execution engine, and we are working to standardize these as part of the ANSI SQL/MED proposal.

In [VLDB-99a], we described the framework we developed to support the wrapper's need to provide cost estimates to the optimizer for each plan it returns. It is essential for the extensibility of the system that wrappers do this; otherwise, either the optimizer would have to be changed for each new data source, or the system would not be able to do cost-based optimization, meaning that performance would suffer dramatically. However, in general the task of constructing a cost model for a new data source is a large and intimidating one. The wrapper writer essentially has to develop their own model of the data source, analyze the query, and determine a cost. The framework we developed makes it easy for wrappers to provide cost information, requires few changes to a conventional bottom-up optimizer, and is easily extensible to a broad range of sources. We provide storage for statistics in the middleware, a basic facility for updating statistics, cost equations for computing selectivities of predicates, and overall cost computation for plans. Wrappers can use these facilities "as-is", tailor them to meet their needs, or override them altogether. We believe that our framework for costing is the first to allow accurate cost estimates for diverse sources within the context of a traditional cost-based optimizer. Experiments demonstrate the importance of cost information in choosing good plans, the flexibility of the framework, the accuracy it allows, and finally, that it works -- the optimizer is able to choose good plans even for complex cross-source queries.

In writing the wrappers mentioned above, we noticed that many real life wrappers needed to cache data for objects returned to the application, so that attributes of those objects could be quickly returned on subsequent method calls. The Web Wrapper, in particular, needed this ability; else the performance became so bad as to be unusable. However, other wrappers also were doing some amount of caching to improve their performance. Rather than force each wrapper writer concerned about performance to build their own cache manager, we decided to provide a simple service that automatically caches all the attributes of an object referenced, and to modify the method dispatch mechanism to exploit the cache. Thus, wrapper writers get caching "for free", dramatically improving performance for those with slow access to data or slow network connections. We investigated ways to allow queries to load the cache, as well, enhancing the performance of applications that use queries to identify the set of objects they need, and then use methods to retrieve data from the objects. We developed a new cache architecture that can be loaded efficiently with query results without flooding out cached objects that are currently of interest to the application. This work is described in detail in [VLDB-99b].

A concrete schema is a description of alternative paths for accessing sets of objects. A concrete schema is critical for efficiently processing queries containing paths, because it greatly expands the set of feasible plans for executing such queries. The expanded set of plans may include ones that are vastly more efficient than those suggested directly by the query, because, e.g., they can take advantage of an index on a collection not explicitly mentioned in the query, or allow a larger piece of the query to be delegated to a single repository. In [RJ-00], we describe a language by which a Garlic database administrator can specify a concrete schema. The work is novel because the more restrictive data models of traditional relational or object-relational databases make concrete schema specifications unnecessary. Our language design strikes a careful balance between

expressive power, simplicity, and implementability. Its syntax and semantics are straightforward, and a database administrator can easily express the specifications that are most likely to lead to large improvements in query execution speed. The middleware query processor can easily determine the specifications applicable to a query, and efficiently exploit the relationships they describe. Specifications are optional: omission of a specification cannot lead to incorrect query results. We built a prototype of this facility, and demonstrated that, at the price of some additional optimization time, dramatic savings in execution time can be achieved.

The result of this work has been a wrapper architecture and a set of support services for wrappers that we believe are the first to meet the goals stated above. The architecture is flexible, allowing data to be queried from sources that may be complex or simple or idiosyncratic. It is productive: wrappers can be written quickly and evolved easily. It is effective, allowing queries access to all data types, as well as to the query processing and data manipulation capabilities of the source. It provides excellent support for query optimization, enabling efficient cross-source queries. We feel that this work represents a significant contribution to the field of heterogeneous data integration.

BIBLIOGRAPHY

[CIKM-95] M. Tresch, A. Luniewski, "An Extensible Classifier for Semi-Structured Documents", In *Proc. On Information and Knowledge Management*, Baltimore; Maryland, November, 1995.

[ODMG] R. G. G. Cattell, "The Object Database Standard -- ODMG-93", Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[RIDE-DOM'95] M. Carey, L. Haas, et al., "Towards Heterogeneous Multimedia Information Systems", In *Proc.of the Intl.Workshop on Research Issues in Data Engineering*, Taiwan, March 1995.

[RJ-00] P. Schwarz, L. Haas, B. Niswonger, F. Ozcan, "Optimizing Queries Containing Path Expressions Using Alternative Logical Access Paths", submitted for publication.

[VLDB-93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, J. Thomas, "The Rufus System: Information Organization for Semi-Structured Data", In *Proc. 21st Int'l Conf. On Very Large Data Bases (VLDB),*Dublin, Ireland, xxx, 1993.

[VLDB-95] M. Tresch, N. Palmer, A. Luniewski, "Type Classification of Semi-Structured Documents", In *Proc. 21st Int'l Conf. On Very Large Data Bases (VLDB),* Zurich, Switzerland, September, 1995.

[VLDB-97] M. Tork Roth, P. Schwarz, "Don't Scrap It, Wrap It: A Wrapper Architecture for Legacy Data Sources", In *Proc. 25th Int'l Conf. on Very Large Data Bases (VLDB),* Athens, Greece, August 1997.

[VLDB-99a] M. Tork Roth, F. Ozcan, L. Haas, "Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System", In *Proc. 27th Int'l Conf. on Very Large Data Bases (VLDB),* Edinburgh, Scotland, September 1999.

[VLDB-99b] L. Haas, D. Kossmann, I. Ursu, "Loading a Cache with Query Results", In *Proc. 27th Int'l Conf. on Very Large Data Bases (VLDB),* Edinburgh, Scotland, September 1999.

An Extensible Classifier for Semi-Structured Documents *

Markus Tresch** Allen Luniewski

IBM Almaden Research Center
650 Harry Road (K55/801)
San Jose, CA 95120, USA
tresch@inf.ethz.ch, luniew@almaden.ibm.com

Abstract

In this paper, we present a vector space classifier for determining the type of semi-structured documents. Our goal was to design a high-performance classifier in terms of accuracy (recall and precision), speed, and flexibility.

The ability to dynamically extend a classifier with user-specific classes is crucial for many applications. Unfortunately, the training data of existing classes is often not available, such that the extended classifier is imprecise as a result.

We focus on this issue. First, we evaluate how to create class abstracts that can be used as training data replacement. Second, we introduce relevance feedback learning strategies to overcoming the remaining classifier flaw.

1 Introduction

Novel networked information services [ODL93], for example the World-Wide Web, offer a huge diversity of information: journal articles, electronic mail, C source code, bug reports, television listings, mail order catalogs, etc. Most of this information is semi-structured. In some cases, the schema of semi-structured information is only partially defined. In other cases, it has a highly variable structure. And in yet other cases, semi-structured information has a well-defined but unknown schema. For instance, RFC822 e-mail follows rules on how the header must be constructed, but the mail body itself is not further restricted.

The first step towards automatic processing of semi-structured documents is *classification*, i.e., to assign an explicit type to them. Thus, a classifier is necessary that explores the implicit structure of such a document and assigns it to one of a set of predefined classes (e.g. document categories or file types) [GRW84, Hoc94].

This class can then be used to apply type-specific methods that, for example, extract values of (normalized and non-normalized) attributes in order to store them in specialized databases. For instance, an object-oriented database might be used for storing complex structured data, a full text database for natural language text, and an image database for pictures.

The Problem. Classifiers are usually created in two steps. First, the class hierarchy must be defined by giving features (distinguishable attributes) for each class. Second, the classifier is trained with a set of typical documents for each class.

However, this basic class hierarchy must be extensible. Users want to define and add specific classes according to their personal purposes. *Extensibility* of a classifier is therefore crucial for many applications. In order to add classes to a classifier, a user must provide training data for the new classes. Training documents of the existing classes must be available too. This "old" training data is necessary because new classes must be trained with data for existing classes as well. As we will formalize later in this paper, if the "old" training data is missing, the extended classifier will be insufficiently trained, and consequently low in recall.

Unfortunately, training data for the basic classes is often not available to a user at the time of extending the classifier. For instance, training data sets are potentially voluminous and/or may be proprietary, which make their distribution with the classifier undesirable.

The Approach. In this paper, we present and evaluate the benefit of a novel two-step approach to overcoming the imprecision of insufficiently trained classifiers due to incomplete or missing training data. Both steps together are required for classifier extension:

1. The first step employes automatic full text indexing techniques to summarize training data into *class abstracts*. These abstracts are used as training data replacements. We present strategies to create abstracts that are much smaller than the training data, but still accurately characterize a class.

2. In the second step, a novel algorithm for iterative classifier learning is presented. This algorithm is based on *relevance feedback*: The classifier provides a human expert with feedback about how confident it is on the classification of a document, and the user gives feedback to the classifier about documents that must be learned in order to improve its accuracy.

*Research partially supported by Wright Laboratories, Wright Patterson AFB, under Grant Number F33615-93-C-1337.

**Present Address: Swiss Federal Institut of Technology, Department of Computer Science, Database Research Group, ETH Zentrum, CH-8092 Zurich, Switzerland.

CIKM '95, Baltimore MD USA
© 1995 ACM 0-89791-812-6/95/11..$3.50

This paper is organized as follows. In Section 2, we review our experimental vector space classifier. In Section 3, we formalize the problem of classifier extensibility and show how to summarize training data into abstracts. In Section 4, we introduce classifier learning algorithms based on relevance feedback. In Section 5, we discuss related work, and conclude in Section 6 with a summary and an outlook.

2 An Experimental Vector Space Classifier

In this section, we introduce an experimental classifier system based on the vector space model [SWY75]. The goal was to build an extremely high-performance classifier, in terms of accuracy (precision and recall), speed, and flexibility for the classification of files by type. UNIX was considered as a sample file system. The classifier examines a document (a UNIX file) and assigns one of a set of predefined classes (UNIX file types). To date, the classifier is able to distinguish the 47 different file types illustrated in Figure 1. Note that the classifier presented here can be generalized to most non-UNIX file systems.

This classifier plays a key role in the Rufus System, where the explicit type of a file – assigned by the classifier – is used to trigger type-dependent extraction algorithms [SLS$^+$93]. Based on this extraction, Rufus supports structured queries that can combine queries against the extracted attributes as well as the original files. In general, a file classifier is necessary for any application that operates on a variety of file types and seeks to take advantage of the (probably hidden) document structure.

Vector space classifiers (VSC) are created for a particular classification task in two steps:

1. *Schema definition.* The features $f_1, \ldots, f_m$ are defined and form the classifier schema. The features span an m-dimensional feature space. In this feature space, each document d can be represented by a vector of the form $v_d = (a_1, \ldots, a_m)$ where a_i gives the value of feature f_i in document d.

2. *Classifier Training* The classifier is trained with training data – a collection of typical documents for each class. The frequency of each feature f_i in all training documents is determined. For each class c_i, a centroid $v_{c_i} = (a_1, \ldots, a_m)$ is computed, whose coefficients a_i are the mean value of the extracted features of all training documents for that class.

2.1 Defining the Classifier's Schema

A feature is some identifiable part of a document that distinguishes between document classes. For example, a feature of LaTeX files is that file names usually have the extension "`.tex`" and that the text frequently contains patterns like "`\begin{...}`". Features can either be boolean or counting. Boolean features simply determine whether or not a feature occured in the document. Counting features determine how often a feature was detected. In the sequel, we assume boolean features only and refer to [TPL94] for a discussion of counting features.

To define the schema of the UNIX file classifier, four different types of features are supported. They can be used to describe patterns that characterize file types:

- A `filenames` pattern is matched against the name of a file;
- A `firstpats` pattern is matched against the first line of a file;
- A `restpats` pattern is matched against any line of a file;
- An `extract` feature specifies that there exists an extraction function, that is, an external procedure like a C program, to determine if the feature is present.

The first three feature types are processed by a pattern matcher. For performance reasons, this is a finite state machine specially built from the classifier schema. Patterns can either be string literals or regular expressions. Regular expressions supported so far are similar to the regular expressions of the UNIX "ed" command. The fourth feature type, `extract`, is a C program that examines files for specific properties. For instance, it can be used to check whether a document is an executable file or a directory.

Any feature can be defined as `must`, which means that its occurrence is mandatory. If such a feature is not present in a given file, the file cannot be a member of that class. Notice that the converse is not true. The presence of a `must` feature does not force a type match.

Example 1: The following figure shows an excerpt of a sample classifier schema, defining classes for file types corresponding to PostScript pictures, LaTeX documents, MH-Folder directories, and Compress files. ◇

```
PostScript {
    filenames {
        "\.ps$" regexp
    }
    firstpats {
        "^%!" regexp must
    }
    restpats {
        "%EndComments"
        "%Creator:"
    }
    abstract {
        affinity 30 5K 10K
        abstr.ps
    }
}

MHFolder {
    extract
    abstract {
        directory
        abstr
    }
}
```

```
LaTeX {
    filenames {
        "\.tex$" regexp
    }
    restpats {
        "\begin{"
         "\end{"
         "{document}"
    }
    abstract {
        affinity 30 5K 10K
        abstr.tex
    }
}

Compress {
    extract
    filenames {
        "\.[zZ]$" regexp
    }
    abstract {
        sample 50 5K 10K
        abstr.Z
    }
}
```

Finding appropriate features for each class is crucial to the accuracy of a classifier [Jam85]. It is difficult to automate this task. For example, to define the 47 classes of the UNIX file classifier, a total of 206 features were carefully specified. In general, there are two complementary approaches to feature selection and analysis:

- an analyzer scans all training documents and proposes features with high distinguishing power;
- an analyzer scans human generated features and identifies those with poor distinguishing power.

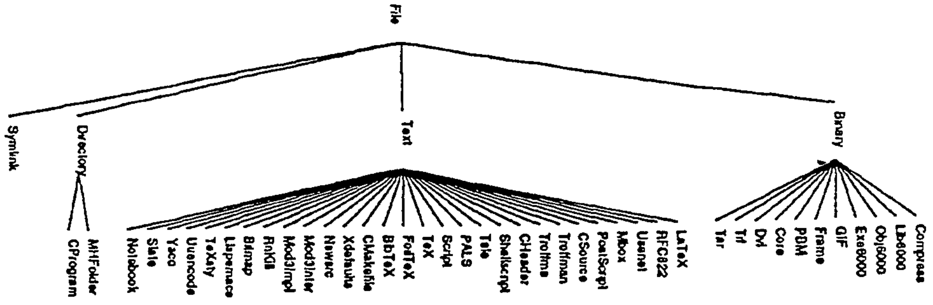


Figure 1: Class hierarchy of the experimental file classifier

In both cases, a human expert is necessary to decide whether some feature must be included although they are not very distinguishing, for instance, those that are the only feature of a top-level class in the hierarchy (TEXT, BINARY, DIRECTORY, SYMLINK). On the other hand, regular expression patterns, for example, may contain an error that cannot be detected and corrected automatically.

There exist several techniques for separating good and bad features. In our system, we use the notion of *distinguishing power* to identify highly distinguishing patterns (cf. [TPL94]).

2.2 Training the Classifier

To train the classifier, a human expert has to provide a reasonable number of documents that are typical of each class. The main issue in training a classifier is: how much training data is required for it to perform well.

Experiments building the classifier with different numbers of training documents showed that a surprisingly small set of training data, namely 10 to 20 documents per class, produces a sufficiently accurate classifier. For example, a classifier built with only one random training document per class has average recall/precision[1] of 0.85/0.94. The same classifier with 5 random training documents has average recall/precision of about 0.95/0.98, with 10 documents about 0.96/0.98, and with 20 documents (~1000 total) 0.97/0.99.

In the experiments discussed in the remainder of this paper, we use (unless stated otherwise) training data sets with an average of 17 documents per class (total ~800 documents = ~26 MBytes). These data sets have randomly been selected as subsets of a large collection of training documents that was gathered by manually classifying thousands of files from our company-wide UNIX file system.

2.3 Finding the Most Similar Centroid

Given a trained classifier with centroids for each class, classification of a document d means finding the "most similar" centroid v_{c_i} and assigning d to that class c_i. A commonly used similarity measure is the *cosine metric* [vR79]. It defines the distance between document d and class centroid c_i by the angle α between the document vector and the centroid vector, that is,

$$\mathrm{sim}(d, c_i) = \cos\alpha$$

A large number of alternative similarity measures are proposed in the literature. For example, [vR79] describes Asymmetric, Dice, Jaccard, Overlap, and Euclidian distance and many more can easily be imagined. However, experiments showed that the best and most reliable results, in terms of recall and precision, are achieved using the cosine as similarity measure (cf. [TPL94] for a detailed discussion, including performance experiments and results).

Furthermore, the cosine metric is intuitive and can be easily computed. For example, on an IBM Workstation RISC System/6000 model 530H, an individual document is classified by the experimental classifier in about 40 milliseconds, on average.

3 Extending the Classifier

We now concentrate on the main focus of this paper, namely extending a VSC with new classes. We assume that – for the afore mentioned reasons – training data is only available for new classes, not for existing ones.

To formalize the error due to adding new classes, consider a classifier with existing classes $c_1, \ldots, c_g$ and existing features $f_1, \ldots, f_j$. Assume this classifier is extended with new classes $c_h, \ldots, c_n$ ($h = g + 1$) and new features $f_k, \ldots, f_m$ ($k = j + 1$). After extension, each coefficient a_{xy} of the *feature-centroid matrix* A of the classifier, shows the value of feature y in the centroid of class x.

The upper-left sub-matrix of A shows the feature-centroid matrix of the classifier before extension. The upper-right sub-matrix of A, call it D, shows the centroid coefficients of the new features for the existing classes. D reflects whether or not training data of existing classes has been used for extension. If full training data were used, D would be "accurate". However, as previously discussed, full training data for existing classes will not be available at the time of extension. Thus, A will be only approximately correct. More precisely, if no training data is available for existing classes, all coefficients of D are zero.

3.1 Abstract Creation

The first step in making a VSC extensible is automatic construction of *class abstracts* (profiles, summaries) from each

[1] The accuracy of a classifier is measured for a particular class C as [Jon71]

$$\mathrm{Recall}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects of } C}$$

$$\mathrm{Precision}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects assigned to } C}.$$

To measure a classifier as a whole, we use the arithmetic mean of recall or precision over all classes. Notice that every object is classified into exactly one class (no unclassified or double classified objects).

class' training data. This is a compromise between no and full training data. In contrast to the training data itself, these abstracts will be generally available and play the role of training data surrogates during extension. For new classes, we assume full training data.

Class abstracts must be significantly smaller than full training data. Though we know from Section 2.2 that 10 to 20 training documents per class are sufficient, this is still a potentially large amount of data (~ 26 MBytes for a classifier with 47 classes in our case). A challenging goal is that the size of an abstract should not exceed 1% of the size of the full training data. At the same time, abstracts should accurately characterize that class. For example, the classifier's performance (accuracy) should reach 95% of the performance of a classifier based on full training data.

The way in which abstracts are created depends on the actual file type. The experimental classifier currently provides four general strategies for building abstracts:

- *Textual style classes.* A simple and fast approach is to scan the training data and build abstracts containing the most frequently used words.

  This strategy does not take into account features that are regular expressions defined over multiple words. Consider for example feature "END_*; regexp" of file type Mod3Impl. This feature says that Modula-3 implementations frequently contain the string literal "END" followed by any number of blanks and a ";".

  The lexical affinity algorithm [MBK91] is more sophisticated and uses word co-occurences. It constructs an abstract from the most resolving word pairs (*n*-tuples, in general).

  This strategy is well suited to most textual document classes that are identified by single string literals or small sequences of literals, e.g. Mod3Impl, Usenet, PostScript, ...

- *Binary style classes.* Binary style classes usually have a special string (magic number) at the beginning of the file's first line (e.g., Lib6000, GIF, Frame, Dvi, Tif). All other features are special C functions that search the file for a magic number. Hence, it is very difficult to define an abstraction strategy that is appropriate for a large variety of binary classes.

  One simple algorithm that works well is random sampling which takes samples from the training data to build an abstract.

  This strategy is mainly used for binary classes, e.g. Compress, Tar, Obj6000. It is also well-suited to some special kinds of textual documents that are identified by long contiguous sequences of words rather than frequently used keywords, e.g., Bitmap, Slate, Tele.

- *Very special classes.* A straight copy strategy creates an abstract as a full copy of the training data.

  This strategy is mainly used for very special document classes, whose files are already small, e.g. symbolic links (Symlink).

- *Directory style classes.* A directory strategy is used for directory structured document classes. It creates an abstract that is itself a directory of abstracts. It recursively applies the appropriate choices from of the strategies above to all files of the original directory, creates abstracts of these files, and stores them in the newly created abstract directory tree.

  This strategy is used, for example, to create abstracts for classes Directory, MHFolder, and CProgram.

The schema file contains special abstract statements to define for each class an abstract strategy, its size, and the abstract's name. As an example, consider the schema definition in the above example. Notice that the VSC does not distinguish whether it uses. abstracts or the "original" training data. Hence, the name of a class abstract must be chosen carefully for document classes with filenames patterns.

3.2 On the Quality of Class Abstracts

To measure the quality of class abstracts, one can train a classifier using only abstracts for all classes and comparing its accuracy with a classifier that was trained with full training data for all classes. Table 1 summarizes several experiments. First, a classifier was built with a schema create by an unexperienced user resulting in low recall, and second, with a classifier based on a much improved schema created by an expert. In each test run, both classifiers were tested on the same test data sets.

Table 1: Full training data vs. abstracts

(a) unimproved schema	*avg. recall*	*avg. precision*
full training data	0.86 (100%)	0.86 (100%)
abstracts only	0.74 (86.1%)	0.80 (93.8%)
(b) improved schema	*avg. recall*	*avg. precision*
full training data	0.95 (100%)	0.96 (100%)
abstracts only	0.93 (98.3%)	0.94 (99.8%)

Table 1(a) shows an average recall/precision of 0.86/0.86, if created with full training data, as opposed to 0.74/0.80 if trained with abstracts only. Hence, using abstracts only for training achieved 86.1% of the recall and 93.8% of the precision of a classifier trained with full training data. The goal of 95% classifier accuracy, if using abstracts instead of full training data, was obviously not yet achieved.

In Table 1(b), we mainly made the features more distinguishing by applying the distinguishing power method [TPL94]. The average recall/precision increased to 0.95/0.96, just from using the optimized schema. The average recall/precision of a classifier created with only abstracts was improved to 0.93/0.94. Notice that we now exceed our goal in terms of abstract quality. A classifier built with abstracts reached over 98% of the accuracy of a classifier built with full training data. Consequently, schema tuning not only dramatically improves the classifier's overall accuracy, but also the relative quality of the abstracts.

Notice that this experiment still does not really reflect classifier extension, where abstracts are used for existing classes only. For new classes, full training data is taken. As formalized with the feature-centroid matrix $\mathcal{A}$, the error due to missing training data – and therefore the use of abstracts – depends on the ratio between the number of existing and added features, as well as the overlapping of the new features with the existing classes. Hence, Table 1 is a "worst case analysis" that will never apply in the extensibility scenario.

4 Relevance Feedback Learning

Class abstracts do not completely eliminate centroid flaws. Even if they did, abstracts are built from (potentially old) training data that may no longer represent the actual user data. Hence, even a well trained classifier must adapt to actual needs.

Consequently, a second step is required that deals with the remaining centroid error and ensures that a classifier is accurate over time. The technology employed here is *relevance feedback*, i.e. we show how a classifier can incrementally learn from a human expert. In particular, we use *vector space modification* to adjust the class centroids. This approach is similar to query modification known from information retrieval systems [Roc71, Ide71, Har92].[2]

Iteratively, a human expert looks at test documents classified by an existing classifier and forms, depending on the chosen learning strategy (see below), two sets of learning documents: $\{D_1, \ldots, D_{n_d}\}$ with centroid $D = \sum_j \frac{D_j}{n_d}$ and $\{E_1, \ldots, E_{n_e}\}$ with centroid $E = \sum_j \frac{E_j}{n_e}$. In each iteration step, better approximations of the optimal class centroids are obtained by moving centroids C towards D and away from E, i.e.,

$$C^{(i+1)} = \alpha C^{(i)} + \beta D - \gamma E \ , \qquad (1)$$

where $C^{(i)}$ is the centroid after the i-th learning step.

α, β, and γ control how far the centroids are moved per learning step. In the literature, experimental values can be found for query modification in information retrieval systems. For example, [SB90] compared different methods and achieved best results with $\alpha = 1.0, \beta = 0.75, \gamma = 0.25$. However, no formal reasons are given for this choice of values.

Constant multipliers did not give stable results in our experiments. Our conjecture is that we were adjusting the centroids too "aggressively" and were, in essence, trying to track variations of features that are inherent in documents of a given class. In other words, this approach failed to take advantage of the fact that the information contained in a centroid increases after each learning iteration.

To overcome this problem, we set α, β, γ based on the *information content* of a centroid. For a particular iteration, let n_c be the number of documents from which centroid C was computed. We define

$$\alpha = 1 - \frac{n_d}{n_c + n_d} + \frac{n_e}{n_c + n_e} \qquad \beta = \frac{n_d}{n_c + n_d} \qquad \gamma = \frac{n_e}{n_c + n_e} \, . \qquad (2)$$

This strategy moves centroid C proportional to the number of underlying training/learning documents. After each learning iteration, n_c increases as $n_c^{(i+1)} = n_c^{(i)} + n_d^{(i)} + n_e^{(i)}$, and therefore, the effect of new learning is reduced over time.

The learning strategy determines which documents the classifier learns from. There are two approaches:

- *Negative Learning Strategy:* The classifier learns from documents a human expert identifies as incorrectly classified, that is, D = documents of class C, but NOT classified as C, and E = documents NOT of class C, but classified as C.
- *Positive Learning Strategy:* The classifier learns from documents a human expert identifies as correctly classified, that is, D = documents of class C, correctly classified as C. E = second closest centroids of documents classified correctly as C. To reinforce the effect, the centroid is not only moved towards correctly classified documents, but also away from second closest centroids.

To illustrate these strategies, initial classifiers were built with one randomly selected file per class. Then, 10 feedback steps were performed, learning 20 documents each time. Disjoint data sets were used for initial training, learning, and testing. Figure 2(a) shows the total number of *learned files* (x-axis) needed to achieve a certain average E-value[3] (y-axis). It clearly shows that a negative feedback strategy (learning from 20 incorrectly classified files) gives the best learning curve, as opposed to a positive strategy (learning from 20 correctly classified files), or learning from any file (any 20 classified files).

However, the really critical factor in the efficiency of a learning algorithm is not the number of documents a classifier has to learn from to achieve a given accuracy improvement, but the number of documents a human expert has to look at (touch). We propose the following definition:

Definition. The *efficiency* of a learning strategy is defined as

$$\text{learning efficiency} = \frac{\Delta\text{E-value}}{\#\text{ of touched files}} \, .$$

Figure 2(b) gives a revised view of the same data, but now uses the number of *touched files* as the x-axis. The negative learning strategy (dashed line), that had the fastest learning curve in Figure 2(a), now requires significantly more test files to be touched by the user, compared to the other strategies. The major problem is finding incorrectly classified files. Remember, our goal is to improve a classifier that already classifies more than 90% of files correctly. Hence, to find 20 incorrectly classified files, a human expert has to touch about 200 files. Since this gets even worse as the classifier gets better, a fast way how to identify incorrectly classified documents is necessary.

4.1 The Confidence Measure

Earlier in this paper, we selected the cosine similarity metric to find the closest centroid. However, independent of which similarity measure is chosen, closeness to a centroid is not a very useful indicator of the classifier's *confidence*. Thus, we introduce the following novel measure that gives feedback on how sure the classifier is about a result.

Definition. The *confidence* of an assignment of document d to class c_i is defined as

$$\text{confidence}(d, c_i) \overset{\text{def}}{=} \frac{\text{sim}(d, c_i) - \text{sim}(d, c_j)}{\text{sim}(d, c_i)}$$

with c_i the closest and c_j the second closest centroid.

The confidence is the ratio of the similarity of the closest and second closest centroid over the similarity of the file and the closest centroid.

[2] A different approach – that is not further considered here – is feature weighting, what relates to term weighting in information retrieval systems. Here, the vector space remains unchanged, but weights are introduced that increase or decrease the importance of certain features.

[3] The E-value [vR79] is a single measure of classifier accuracy that combines and equally weights both, recall and precision, as E-value = 1 − (2 Precision Recall)/(Precision + Recall).

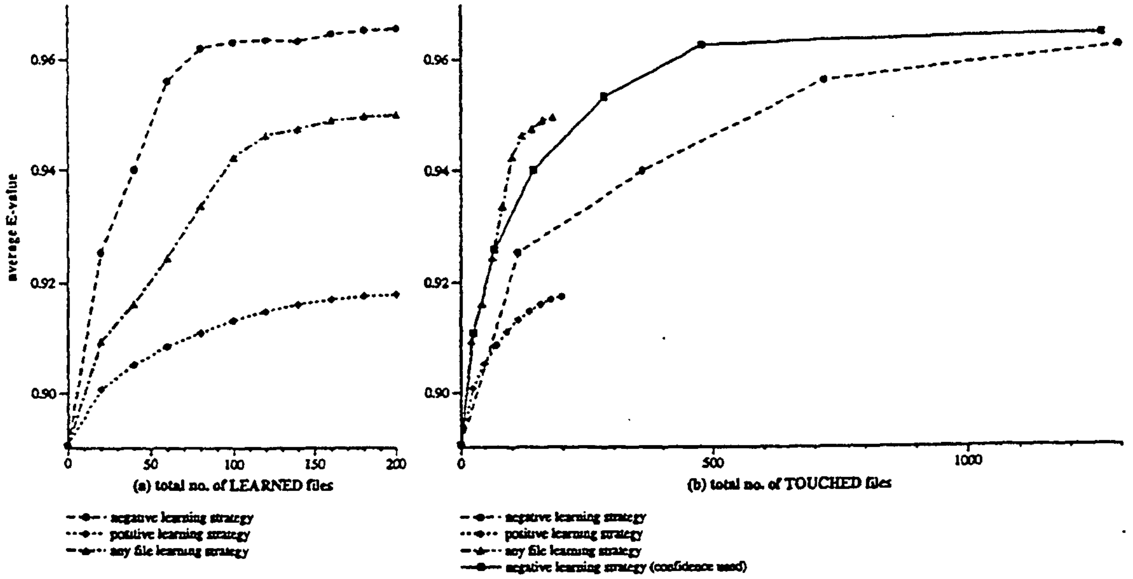


Figure 2: Negative vs. positive feedback strategies

A classifier can use the confidence measure to alert a human expert whenever a classification is below a given *confidence threshold* Θ. Figure 3 illustrates that the higher the confidence value, the higher the probability that the classifier classified the file correctly.

Assume a given confidence threshold Θ (vertical line), such that the user has to approve the classification if a file is classified with a confidence below that threshold.

The dotted line shows the percentage of test files for which the assumption is true that they are classified correctly if classified with a confidence above threshold Θ and classified incorrectly otherwise. If, for example, the threshold Θ is set to 0.1, then about 94% are classified correctly if their confidence is above 0.1 and incorrectly otherwise (see dotted line hitting threshold).

The solid line shows the percentage of test files that were classified with a confidence below Θ. With Θ = 0.1, about 10% of the files are presented to the user for checking (see solid line hitting the threshold). These were shown to a human expert.

Finally, the dashed line shows the percentage of test files that were classified correctly even though they have a confidence below threshold Θ. These are the files where the classifier "annoyed" the user for no good reason. With Θ = 0.1, only 30% of the presented files were actually classified correctly (see dashed line hitting the threshold). Thus, using the confidence measure, a user had to touch 10% of all files, of which in fact 70% were classified incorrectly. The classifier's overall recall could therefore be improved by 7% without bothering the user too much.

Θ = 0.1 provides a maximum accuracy (dotted line) while providing a reasonable number of files for the user's consideration while maintaining a modest "annoyance" level.

4.2 Mutual Feedback Learning Algorithm

Low confidence is a good sign of incorrect classification. Consequently, we can use this measure to find incorrectly classified documents in a relevance feedback learning strategy.

At recall 0.90, about 70% of the least confidently classified files are indeed incorrectly classified. I.e., to find 20 incorrectly classified files to learn from, a human expert has, on average, to touch only the 29 least confidently classified files.

The result is illustrated in Figure 2(b): The solid curve shows the negative learning strategy that uses the confidence measure to find incorrectly classified files. Obviously, it has a much better learning efficiency, because the number of touched files is reduced dramatically.

The following classifier learning algorithm is based on bidirectional *mutual feedback*: The classifier provides feedback to a human expert in terms of how confident it is about a classification, and the user gives feedback to the classifier in terms of how particular files should be classified.

```
while human expert is willing to approve documents do
    classify $N_0$ documents using current classifier;
    learning set $L := \emptyset$;
    for $i = 1$ to $N_1$ do
        ask human expert to approve $i$th least
        confidently classified document $d_i$;
        if $d_i$ classified incorrectly then
            add $d_i$ to $L$;
    end;
    learn all documents in $L$, using negative
    learning strategy;
end
```

Choosing the number of test documents per iteration is not critical. It can be set fairly high, e.g., $N_0 = 1000$, or to even all the documents that are available, because classifying test documents and determining the confidence does not require user interaction.

Notice that this algorithm requires the user to touch exactly the same number of documents each iteration, namely N_1. As the classifier gets better, fewer incorrectly classified files are found among the N_1 least confidently classified ones, and the number of learned documents decreases over time.

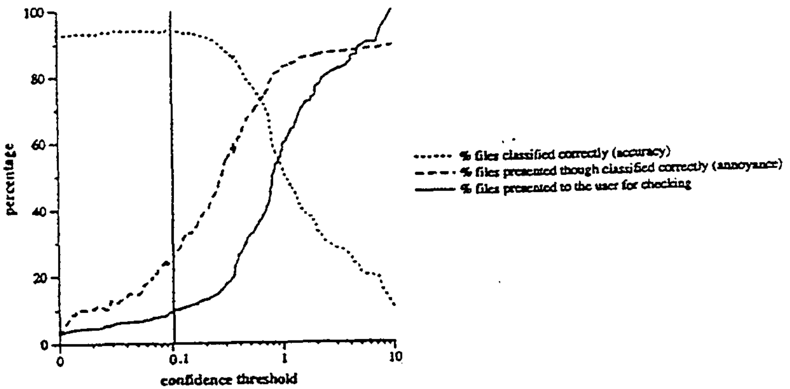


Figure 3: Feedback from the confidence measure

The number of touched documents per step determines the steepness of the learning curve and the finally achieved E-value. The higher the number of touched documents per step, the flatter is the learning curve. However, the higher is the achieved accuracy because a higher variety of documents are included into the learning process. Figure 4 illustrates this behavior of the algorithm for $N_1 = 10, 25, 50, 75$.

The mutual feedback learning algorithm is tuned to minimize the user's interaction (in terms of touching test files), while at the same time, it achieves highest accuracy.

5 Comparison of Related Classifier Technologies

There are diverse technologies for building classifiers. *Discriminant analysis* (linear or quadratic) is a well known basic statistical classification approach [Jam85]. *Decision tables* are very simple and determine according to table entries what class to assign to an object. The UNIX "file" command is an example of a decision table based file classifier. It scans the /etc/magic file, the decision table, for matching values and returns a corresponding string that describes the file type. *Decision tree classifiers* construct a tree from training data, where leaves indicate classes and nodes indicate some tests to be carried out. CART [BFOS84] or C4.5 [Qui93] are well known examples of generic decision tree classifiers. *Rule based classifiers* create rules from training data. R-MINI [Hon94] is an example of a rule based classifier that generates disjunctive normal form (DNF) rules. Both, decision tree and rule classifiers, usually apply pruning heuristics to keep these trees/rules in some minimal form.

To rate our experimental vector space classifier, we built alternative file classifiers using quadratic discriminant analysis, decision tables, the decision tree system C4.5, and the rule generation approach R-MINI. Our experience can be summarized as follows:

Speed. Training and classification using quadratic discriminant analysis is very slow because extensive computations must be performed. The other classifier technologies provide fast training and classification with performance comparable to a vector space classifier.

Accuracy. Quadratic discriminant analysis and decision tables did not achieve our accuracy requirements. They had error rates up to 30%. The other classifier technologies had much lower error rates. The C4.5 file classifier misclassified from 1.1 to 5.8% of files. The CART file classifier showed error rates from 3.2 to 4.7%. The R-MINI file classifier showed error rates from 0.9 to 5.5%. The vector space classifier had 1.1 to 3.1% error rates. Hence, all three technologies have approximately the same range of errors.

Extensibility. The lack of extensibility of discriminant analysis, decision table/tree and rule classifiers is the most dramatic difference. Extending these classifiers with new user-specific classes demands rebuilding the whole system (tables, trees, or rules) from scratch, that is, it requires complete reconstruction of the classifier.

However, creation of abstracts is independent of a particular classifier technology because it is performed before classifier training. Thus, it is applicable to most other technologies that use feature extraction to learn from training data, including decision table/tree and rule classifier.

Relevance feedback has been thoroughly investigated for information retrieval systems, e.g. in SMART [Roc71, Ide71]. To our knowledge, there is no work using relevance feedback techniques to improve classifiers. The mutual relevance feedback algorithm presented in this paper slightly adjusts centroids. It is specific to the vector space model. Transferring this algorithm to decision table/space classifiers is not possible, according to the above mentioned limitations in extensibility.

6 Conclusion and Outlook

High accuracy and dynamic extensibility are the primary criteria for any classifier. The experimental VSC for files presented in this paper fulfills both requirements.

The file classifier can be seen as a component of object, text, and image database management systems. The classifier can also provide useful services in a next-generation operating system environment. Consider for instance a file system backup procedure that uses the classifier to select file-type-specific backup policies or compression/encryption methods.

Additional experiments have been conducted that uses the classifier for language and subject classification. Whereas language classification showed encouraging results, this technology has its limitations for subject classification. The reason is that the classifier works mainly by syntactical exploration of the schema, but subject classification must take into account the semantics of a document.

We are currently working on the classification of structurally nested documents. A file classifier is being developed that is, for example, able to recognize Postscript pictures in electronic mail or C language source code in natural text documents.

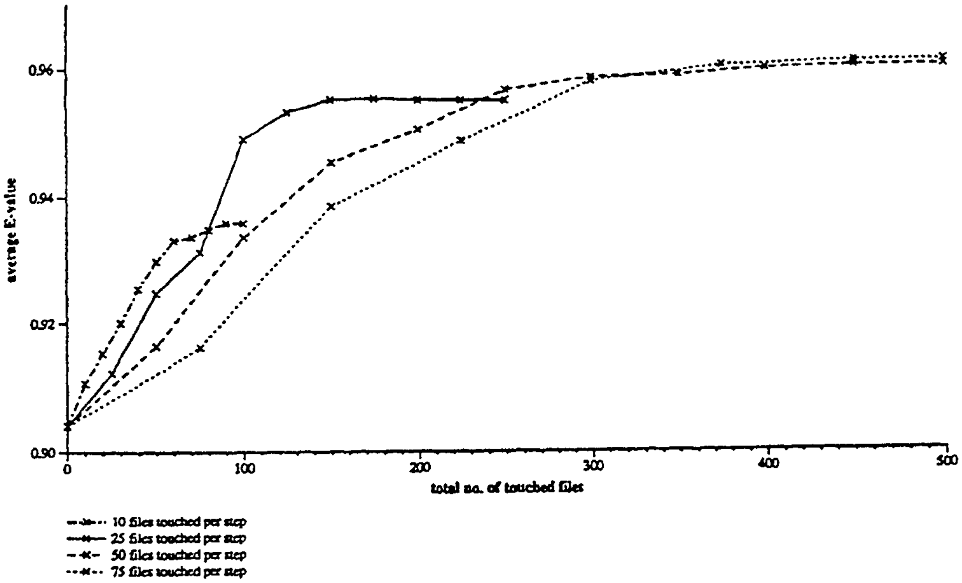


Figure 4: Mutual feedback strategy

References

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, CA, 1984.

[GRW84] A. Griffiths, L.A. Robinson, and P. Willett. Hierarchic agglomerative clustering methods for automatic document classification. *Journal of Documentation*, 40(3), September 1984.

[Har92] D. Harmann. Relevance feedback revisited. In *Proc. 15th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, Copenhagen, Denmark, June 1992.

[Hoc94] R. Hoch. Using IR techniques for text classification. In *Proc. 17th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, Dublin, Ireland, July 1994. Springer.

[Hon94] S.J. Hong. R-MINI: A heuristic algorithm for generating minimal rules from examples. In *Proc. of PRICAI-94*, August 1994.

[Ide71] E. Ide. New experiments in relevance feedback. In G. Salton, editor, *The SMART retrieval system.* Prentice Hall, Englewood Cliffs, NJ, 1971.

[Jam85] M. James. *Classification Algorithms.* John Wiley & Sons, New York, 1985.

[Jon71] K. S. Jones. *Automatic Keyword Classification for Information Retrieval.* Archon Books, London, 1971.

[MBK91] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatic constructing software libraries. *IEEE Trans. on Software Engineering*, 17(8), August 1991

[ODL93] K. Obraczka, P.B. Danzig, and S.-H. Li. Internet resource discovery services. *IEEE Computer*, 26(9), September 1993.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, San Mateo, CA, 1993.

[Roc71] J.J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The SMART retrieval system.* Prentice Hall, Englewood Cliffs, NJ, 1971.

[SB90] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society of Information Science*, 41(4), 1990.

[SLS$^+$93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proc. 19th Int'l Conf. on Very Large Data Bases (VLDB)*, Dublin, Irland, August 1993.

[SWY75] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11), November 1975.

[TPL94] M. Tresch, N. Palmer, and A. Luniewski. Type classification of semi-structured documents. In *Proc. 21th Int'l Conf. on Very Large Data Bases (VLDB)*, Zurich, Switzerland, September 1995.

[vR79] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 1979

Towards Heterogeneous Multimedia Information Systems: The Garlic Approach

M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski,
W. Niblack, D. Petkovic, J. Thomas, J. H. Williams and E. L. Wimmers

IBM Almaden Research Center, San Jose, CA 95120

Abstract:

We provide an overview of the Garlic[1] project, a new project at the IBM Almaden Research Center. The goal of this project is to develop a system and associated tools for the management of large quantities of heterogeneous multimedia information. Garlic permits traditional and multimedia data to be stored in a variety of existing data repositories, including databases, files, text managers, image managers, video servers and so on; the data is seen through a unified schema expressed in an object-oriented data model and can be queried and manipulated using an object-oriented dialect of SQL, perhaps through an advanced query/browser tool that we are also developing. The Garlic architecture is designed to be extensible to new kinds of data repositories, and access efficiency is addressed via a "middleware" query processor that uses database query optimization techniques to exploit the native associative search capabilities of the underlying data repositories.

1: Introduction

In recent years, the problem of heterogeneous distributed data management has attracted a great deal of attention, both from the database systems research community and from developers of commercial database systems. This is due to the fact that improvements in communication technologies, such as affordable high-speed lines, broad deployment of LAN technology, and availability of standardized protocols, have made it feasible to connect historically separate data systems. In commercial environments, such data systems typically include relational databases from various vendors (e.g., IBM, Oracle, Sybase), older, non-relational databases of various genres (e.g., IMS databases), and record-based file systems (e.g., VSAM). Unfortunately, simply connecting all of the related systems does not solve the problem of writing applications that require access to enterprise data from several of them. Though the systems are connected, the data is still represented by different data models, meaning that the programmer must use different access interfaces to get at the data, and must worry about such details as locating the desired data, optimizing access to the different data systems, and managing the transactional consistency of any updates performed.

For traditional business data, solutions are becoming available to some of the problems mentioned above. However, in both commercial enterprises (e.g., retailing or insurance) and in more specialized environments (e.g., health care or engineering design), non-traditional, multimedia data is assuming an increasingly central role. Further, these new types of data are needed for applications (e.g., catalog production, patient records) which also involve traditional, record-oriented data. In both commercial and more specialized environments, the existing multimedia data often resides in file-based data systems that provide media-specific capabilities for searching, storing, and delivering the (often large) multimedia data items. This increased heterogeneity of systems and data types significantly adds to the problems faced by application developers and end-users.

If access to heterogeneous multimedia data is to become common, it will be necessary to develop uniform interfaces for providing location, network, and data model transparency for application developers. Otherwise, interface heterogeneity will inhibit the deployment of large-scale multimedia information systems. Providing uniform access to heterogeneous multimedia data presents a number of new challenges and system requirements. First, both the user and application programming interfaces to the data must be flexible enough to support the new user interaction models that are characteristic when dealing with multimedia data. These include visual query formation and support for both navigation-based querying and similarity queries. Secondly, in order for multimedia informa-

[1] Garlic is not an acronym. Most members of the team really like garlic, and enjoy our laboratory's proximity to the Gilroy garlic fields!

tion systems to be scalable, they must provide convenient mechanisms for integrating additional (new or legacy) data sources and data collections into the system and for smoothly extending the system's querying capabilities to cover such newly integrated data sources.

The goal of the Garlic project at IBM's Almaden Research Center is to build a multimedia information system (MMIS) capable of integrating data that resides in different database systems as well as in a variety of non-database data servers. This integration must be enabled while maintaining the independence of the data servers, and without creating copies of their data. "Multimedia" should be interpreted very broadly to mean not only images, video, audio, but also text and application specific types of data (CAD drawings, medical objects, maps, ...). Since much of this data is naturally modeled by objects, Garlic provides an object-oriented data model that allows data from various data servers to be represented uniformly. This data model is complemented by an object-oriented query language (an object-extended dialect of SQL) and supported by a "middleware" layer of query processing and data access software that presents the object-oriented schema to applications, interprets object queries and updates, creates execution plans for sending pieces of queries to the appropriate data servers, and assembles query results for delivery back to the applications. A significant focus of the project is the provision of support for "intelligent" data servers, i.e., servers that provide media-specific indexing and query capabilities. It is hoped that the Garlic approach to unifying diverse data sources will significantly simplify application development, make end-user data exploration easier, and simplify the problem of integrating new data sources to support the deployment of large-scale, multimedia applications.

The remainder of this paper provides a general introduction to the Garlic project, the approach being taken, and some of the key research issues in this area. The paper begins with a discussion of related work in Section 2. Section 3 provides a general overview of the project, describing the architecture of the system and the role of each of its components. Section 4 presents the Garlic data model, with Section 5 focusing on Garlic's query language and query processing challenges. Finally, Section 6 touches on Garlic's application and end user interfaces. Section 7 concludes with a discussion of the current status of the project and our plans for the immediate future.

2: Related Work

Since the Garlic project is focused on the design and implementation of a heterogeneous multimedia information system based on database concepts and technology, there are two areas where significant related work exists: heterogeneous databases and multimedia systems.

2.1: Heterogeneous Databases

The area of heterogeneous distributed database systems (also known as multidatabase systems) has been the focus of a significant level of research activity over the past five years. Early work in this area actually dates back much further, to projects such as the Multibase effort at CCA [4], but improved communication technologies and resulting commercial demands have led to a recent resurgence of work in the area. Much of the research falls into one of three broad categories: providing uniform access to data stored in multiple databases that involve several different data models; handling similarities and differences between data representations, semantics, and stored values when the same (or similar) information about a given real-world entity is stored in multiple databases; and supporting transactions in heterogeneous distributed database systems in the face of incompatible concurrency control subsystems and/or uncooperative transaction managers. A good survey of the relevant work can be found in [5]. In the commercial realm, products now exist for providing uniform access to data in multiple databases, relational and otherwise, and to structured files, usually through the provision of a unified relational schema. Such "middleware" products are one of the fastest-growing segments of the database market.

Garlic's use of an object-oriented data model to integrate multiple databases is not new, as models with object-oriented features have been employed in projects such as [4], [6], [8] and others. However, a number of the technical aspects of our approach are quite novel, and again, the intended range of target data repositories and data types is much broader. The TSIMMIS project at Stanford University also strives to integrate a broad range of repositories, but using a specially developed Object Exchange Model [26]. What distinguishes the Garlic project from the aforementioned efforts is its focus on providing an object-oriented view of data residing not only in databases and record-based files, but also in a wide variety of media-specific data repositories with specialized search facilities. With the exception of the Papyrus [6] and Pegasus [7] projects at HP Labs, we are aware of no other efforts that have tried to address the problems involved in supporting heterogeneous, multimedia applications.

2.2: Multimedia Systems

The multimedia area today is expanding at an extremely rapid pace, and the area is extremely broad, including work on hypermedia systems, specialized servers with stringent quality of service requirements (e.g., video servers), image and document management tools, interactive games, structured presentations involving mixtures of text, imagery, audio and video data, scripting languages to support timed and synchronized playback of multimedia presentations, and so forth. This is most evident in the personal computer industry, where a large number of small-scale multimedia software packages and products have emerged due to the availability and affordability of CD-ROM technology. In the information systems industry, despite the strong demand for multimedia technologies, things are moving more slowly. In particular, work on multimedia information systems is still in its infancy, with many problems remaining to be solved to combine multimedia data with more traditional, record-oriented data and support associative retrieval and data-intensive multimedia applications [9].

Most work to date on supporting associative retrieval of multimedia data has focused on retrieval methods for specific data types. For example, years of research have produced a solid technology base for content-based retrieval of documents through the use of various text indexing and search techniques [3]. Similarly, simple spatial searches are well-supported by today's geographic information systems ([25], e.g.), and work in the area of image processing has led to systems such as QBIC [1] and Photobook [2] in which images can be recalled based on features such as the color, texture, and shape of scenes and user-identified objects. However, with the exception of simple approaches like attaching attributes to spatial objects, or associating user-provided keywords with images, these component search technologies remain largely isolated from one another. The only notable exception to this is the approach being taken by systems such as the Illustra object-relational DBMS [10], where media-specific class libraries (which Illustra calls DataBlades^{TM}) are provided in order to allow multimedia data to be stored in and managed by the DBMS. Garlic differs in that it aims to leverage existing intelligent repositories, such as text and image management systems, rather than requiring all multimedia data to be stored within and controlled by the DBMS. Our belief is that Garlic's open approach will enable it to take advantage of continuing advances in multimedia storage and search technology. It should also be more effective for legacy environments, where multimedia data collections (such as documents or image libraries) and business data already exist in forms that cannot simply be discarded or migrated into a new DBMS.

3: Garlic Overview

Loosely speaking, the goal of the Garlic project is to provide applications and users with the benefits of a database with a schema -- similar to what an object-oriented or object-relational database system might provide -- but without actually storing (at least the bulk of) the data within the Garlic system proper. Viewed from above, then, Garlic looks rather like a DBMS with an object-oriented schema. Internally, however, Garlic looks very different, with the data that makes up its database being scattered across a number of different data sources. To achieve the former view from the latter reality, Garlic provides two forms of "glue": an object-oriented data model and the ability to store additional complex objects to augment the data in the underlying data sources.

3.1: The Big Picture

Figure 1 depicts the overall architecture of the Garlic system. At the leaves of the figure are a number of data repositories containing the data that Garlic is intended to integrate. Examples of reasonable data repositories include relational and non-relational database systems, file systems, document managers, image managers, and video servers. Above each repository in the figure is a repository wrapper, which translates information about data types and collections (i.e., schemas) and data access and manipulation requests (i.e., queries) between Garlic's internal protocols and that repository's native protocols. Information about the unified Garlic schema, as well as certain translation-related information needed by the various data repositories, is maintained in the Garlic metadata repository. The other repository shown in the figure is the Garlic complex object repository. This repository is used to hold the complex objects that most Garlic applications will need for "gluing" together the underlying data in new and useful ways. Complex objects will be needed to integrate multimedia data with legacy data in situations where the legacy data cannot be changed, and as a place to attach methods to implement new behavior. For example, in an auto insurance application, Garlic complex objects could be used to link the images of a damaged car (stored in an image-specific repository) together with an accident report (stored in a document management system) and a customer's claim and policy records (legacy data residing in a relational database) in order to form a "claim folder" object to be dealt with by an insurance agent.

Query processing and data manipulation services are provided by the Garlic query services and runtime system component shown in Figure 1. This component presents Garlic applications with a unified, object-oriented view of the contents of a Garlic database and processes users' and applications' queries, updates and method invocation requests against this data; queries are expressed in an object-oriented extension of the SQL query language. This component will also be responsible for dealing with transaction management issues. In the near term, Garlic will not provide any guarantees about consistency of legacy data that is operated on by legacy (as well as Garlic) applications. The new Exotica project at Almaden [27] is examining a combination of workflow with transactions which may offer some help for environments such as Garlic's.

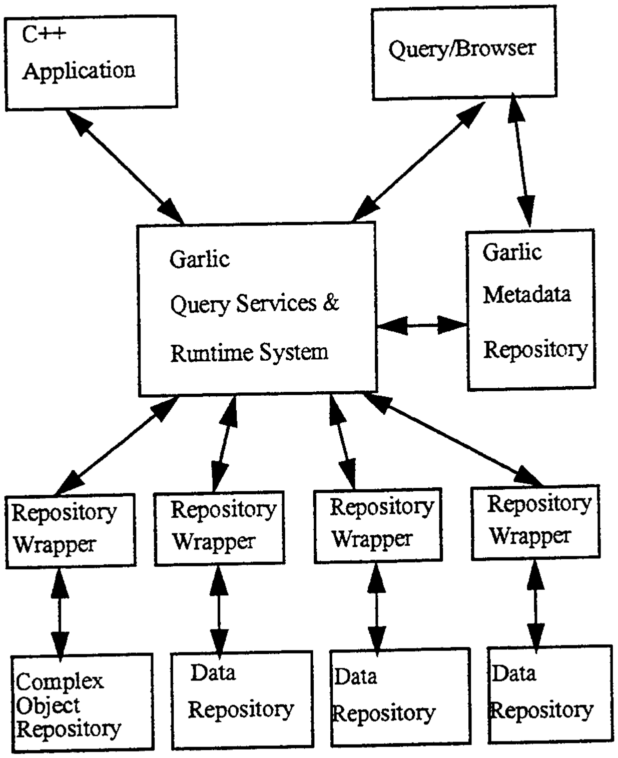


Figure 1.Garlic System Architecture

Finally, Garlic applications interact with the query services and runtime system through Garlic's object query language and a C++ application programming interface (API). Many applications will do this statically, in which case the Garlic schema will be presented to the application via a set of C++ classes that act as "surrogates" for the corresponding classes of the actual Garlic schema. Certain applications may require more dynamic access to the data, in which case they will use a portion of the C++ API that provides dynamic access to information about the types and objects contained in a Garlic database and that enables objects to be manipulated without *a priori* (i.e., compiled) knowledge of the database schema. A particularly important example of such a dynamic application is the Garlic query/browser. This component of Garlic will provide end users of the system with a friendly, graphical interface that supports interactive browsing, navigation, and querying of the contents of Garlic databases.

3.2: Repositories and Databases

As mentioned previously, the purpose of the Garlic system is to integrate and unify data managed by multiple, disparate data sources. Thus, virtually all Garlic databases will involve data, most likely including legacy data, that resides in several data repositories. Up to this point, we have been using the term "data repository" loosely, referring both to a particular kind of data management software (e.g., the DB2 C/S database system, the PixTex/EFS document manager, the QBIC image management library, etc.) and to the collections of data that it manages. However, to explain how we envision Garlic databases being created and used, it is useful to distinguish between the notion of a *repository type*, which is a particular kind of data management software, and a *repository instance*, which is a particular data collection that it manages. We will also refer to a *repository manager*, which is an instance of a repository type (for example, a particular DB2 C/S installation). In order for Garlic to provide access to data that resides in a given repository manager, someone must have written a repository wrapper for that repository type. The Garlic architecture is designed to be extensible in this dimension, and we plan to provide tools to ease the task of writing such wrappers; we will also write such wrappers ourselves for a set of interesting repository types. Given the existence of a wrapper for a repository type of interest, it is then possible to create Garlic databases that include one or more repository instances that are managed by repository managers of that repository type.

The contents of a Garlic database, as mentioned earlier, are presented to applications and to end users as a global schema expressed in terms of the Garlic data model. The Garlic data model, discussed further in Section 4, is based on a proposed object-oriented data model standard [11]. The global schema is in turn the integration of a number of local schemas, one per repository instance, that describe the data contents of each repository instance (including the Garlic complex object repository) that contributes data to the given Garlic database. Each of the local schemas, which are referred to in Garlic as wrapper schemas, is expressed in terms of the Garlic data model as well.[2] Enabling access to the contents of a repository instance through a Garlic database involves identifying the data types and collections that are to be visible as part of the Garlic database and then writing a wrapper schema that describes the target data in GDL (the Garlic Data Language, the syntactic form of the Garlic data model), as well as providing any code required to implement the types' behavior (normally this will form a thin layer on top of the repository type's programming interface). For important repository types that support type systems of their own, e.g., a relational DBMS such as DB2 C/S, we expect that automated schema mapping tools will be provided to assist Garlic database administrators (DBAs) with the task of defining a Garlic wrapper schema to represent

[2] It should be noted that, in the simplest of cases, the global schema may be nothing more than the union of the local schemas. However, it is more likely that the global schema will also involve views that serve to redefine, reshape, and hide some of the data definitions found in the underlying local schemas (see Section 4).

the native data types and collections that they wish to export from a given repository instance.

4: The Garlic Data Model

Our goal in Garlic is to understand the issues associated with providing an integrated, object-oriented view of data from disparate sources, and not to invent yet another object-oriented data model, so we have adopted the ODMG-93 object model [11] as a starting point for the Garlic data model and the syntax of the ODMG-93 object definition language, ODL, as a base for the Garlic Data Language, GDL. Some aspects of the Garlic data model differ from those of the ODMG-93 model, however, as Garlic's heterogeneous environment poses certain problems not found in the logically centralized, OODB environment with which the ODMG-93 standardization effort is concerned.

In the ODMG-93 standard, the fundamental building blocks of the data model are *objects* and *values*. Each object has an identity that uniquely denotes the object, thus enabling the sharing of objects (by reference). Objects are strongly typed, and object types are expressed in the data model in terms of object *interfaces* (as distinct from *implementations*). The description of an object's interface includes the attributes, relationships, and methods that are characteristic of all objects that adhere to the interface. The model also supports an *inheritance* mechanism by which a new interface can be derived from one or more existing interfaces. The derived interface inherits all of the attributes, relationships, and methods of the interfaces from which it is derived, making the derived interface a subtype of those interfaces.

Object identity: The ODMG-93 data model defines object identity in the traditional strong manner, guaranteeing that an object's identity be both unique and immutable. Unfortunately, some repositories, such as relational databases, do not provide this strong notion of identity for the data items that they manage. To enable such data items to be modeled as objects in a Garlic database, references in the Garlic data model are based on a notion that we call *weak identity*; this is simply a means of denoting an object uniquely but not necessarily immutably within the scope of a Garlic database. An object's weak identity is formed by concatenating a token that designates the object's implementation with an implementation-specific unique key. For instance, to view a table in a relational database as a collection of objects in Garlic, the identity of the object corresponding to a relational tuple could be formed by pairing an implementation identifier that (indirectly) specifies the tuple's source table together with the values of the key field(s) of the tuple. An important consequence of weak identity is that it allows Garlic to treat data items for which only weak identification is possible as objects without requiring that proxy (or surrogate) objects be maintained within Garlic for each such data item. Although some systems have employed a proxy-based approach to heterogeneity in object bases (e.g., [8]), we felt that such an approach would cause serious problems in terms of practicality (both space and the cost of maintaining consistency are at issue) and efficiency of access for large legacy databases.

Legacy references: Another problem related to identity and object references in Garlic is the fact that legacy data can contain legacy references. For example, many foreign keys in a relational database are essentially object references, and we would like to show them as such in the portions of the Garlic schema that correspond to the underlying relational data. Similarly, in a repository that stores and indexes documents using a traditional file system for document storage, file names for documents are essentially references to document objects, and they should be shown as such in the Garlic schema. Unfortunately, in both cases, the underlying repository represents and manipulates these legacy references in its own, repository-specific manner. Clearly, to provide access to legacy databases, Garlic cannot require such reference attributes to be converted into and stored using Garlic's full weak identifier format; in fact, Garlic must be careful not to attempt to store long-form references into the reference attributes of legacy data objects. The Garlic data model addresses this problem by providing the notion of an *implementation-constrained reference*, which is a reference that can only denote objects of a specified implementation. Returning to the example of a foreign key, a relational tuple that references another tuple by means of a foreign key cannot reference any arbitrary Garlic object that has the appropriate interface; rather, it can only reference tuples in the specified target table within the same relational database. Thus, when introducing an implementation of a given interface, Garlic supports the association of such implementation constraints with the reference attributes mentioned in the interface. Repository wrappers (see Figure 1) are responsible for converting such reference values between Garlic's object reference format and their repository-specific short forms as needed. Attempts to violate such constraints are detected by the relevant repository wrapper and result in a runtime error.[3]

Extensions: The Garlic data model extends the concepts of the ODMG-93 object model in three significant ways. The first is the degree of support for alternative

[3] It should be noted that while most aspects of the Garlic data model are amenable to static type-checking, this solution is not. However, when a new interface or implementation with the potential to lead to such runtime errors is introduced into the schema, Garlic can provide a warning and even (optionally) prohibit the change.

implementations of interfaces, the second is related to type system flexibility, and the third is an object-appropriate view definition facility.

Garlic makes a sharp distinction between an interface and its implementations. The type of an object is determined solely by its interface, and any number of implementations of a given interface are permitted. It is quite possible that several repositories may offer alternative implementations of an important multimedia data type (e.g., text or image). Garlic supports both the notion of a type extent, which is the set of all instances of a given interface, and an implementation extent, which is the set of all instances managed by a given implementation of an interface of interest.

In terms of type system flexibility, Garlic extends the ODMG-93 model with the concept of conformity [12]. One interface is said to conform to another if the former defines a subtype of the latter under the standard definition of subtyping (including contravariance in the argument positions) -- even if the former interface was not derived from the latter interface using the data model's explicit inheritance mechanism. This notion results in an implicitly-specified type lattice that can provide additional flexibility when independently-defined schemas are merged later, which is clearly an important consideration for Garlic. However, to be compatible wtih ODMG-93 semantics, conformity in Garlic is provided as an option. The definer of an interface A may specify, when using the name of an interface B, whether that use of B should be restricted to mean instances of B and its explicitly derived subtypes, or whether any interface that conforms to interface B is acceptable. This is indicated by saying *conforms(B)* instead of *B* in the relevant part of the definition of A.

Object-Centered Views: The most significant extension that Garlic makes to the ODMG-93 data model is the notion of views. In Garlic, the primary purpose of a view is to enhance (extend, simplify, or reshape) a set of underlying Garlic objects, usually by adding or hiding some of their attributes and/or methods. Garlic employs the notion of an *object-centered view* for this purpose. An object-centered view defines a new interface, together with an implementation of the new interface (usually written using Garlic's object query language), that is based on an existing interface that the view definer wishes to enhance. The existing interface is said to be the *center* of the view, as each element of the view is an enhancement of an object instance whose type is given by the center interface. In the global schema, the view assumes a role similar to a type extent, as its elements can be queried and enumerated just like the objects in a type extent. The elements of a view are considered to be objects, and the identity of each object in a view is derived from the identity of the center object that it enhances. In particular, the identity of an object in a view consists of the identity of its center object prepended with an implementation identifier that indicates the view from which the object was obtained; this allows Garlic to properly handle references to view objects when presented with them later. The methods available on the objects in an object-centered view are selected (or additionally provided) by the view definer, and a view can optionally be positioned in the type hierarchy through explicit placement when the view's interface is defined.

5: Queries in Garlic

Given the schema for a Garlic database that a user or application wishes to utilize, Garlic provides access to the database through a high-level query language. Since the data model of Garlic is object-oriented, and since SQL is the dominant query language today for database application and tool builders, the query language of Garlic is an object-oriented extension of SQL. To accommodate the object-oriented nature of the Garlic data model, Garlic extends SQL with additional constructs for traversing paths composed of inter-object relationships, for querying and materializing collection-valued attributes of objects, and for invoking methods within queries. These object-oriented SQL extensions are similar to extensions provided in various other recent object query language proposals (e.g., [13], [14]), including the ongoing efforts of the SQL-3 committee [15].

Since the Garlic query language is intended for querying databases that contain data in a variety of repositories, including multimedia repositories with associative search capabilities, Garlic's SQL extensions must also take the needs of such repositories into account. Many of their needs can be accommodated simply through the use of the query language's object extensions, e.g., by making use of methods in query predicates and target lists. However, the search facilities provided by repositories that manage multimedia data types such as text and images are often based on a somewhat different query model than the one used in most database systems -- rather than requesting the retrieval of every data item where a given predicate is true, text and image queries commonly request the ordered retrieval of the top N (or all) data items that come "close" (perhaps within a certain threshold) to matching a given predicate. An image-oriented example that could be satisfied using the search capabilities of the QBIC system [1] would be "find all reddish images that contain a yellowish, circle-like object in the middle". QBIC would respond to this query by returning a list of all images that approximately match the specified predicate, and would rank-order the results by their closeness to the query predicate, as measured by an appropriate similarity function.

Integrating approximate match query semantics with more traditional (exact match) database query semantics is an interesting problem, and we are currently developing a set of syntactic and semantic SQL extensions to support queries that involve both exact and approximate search criteria. This work involves introducing into SQL the notion of graded sets (the ordering of which can be modeled using lists in the Garlic data model). In such sets, each object is assigned a number between 0 and 1 for each atomic predicate; this number represents the degree to which the object fulfills the predicate, with 1 representing a perfect match. Boolean combinations of predicates can then be handled using the rules for combining predicates in fuzzy logic [16]. To enable query writers to specify the desired semantics, the syntax of SQL is extended to permit the specification of the number of matching results to be returned and whether or not rank-ordering (rather than an attribute-based sort order, or an arbitrary order) is desired for the query's result set.

5.1: Query Processing

Garlic queries may span a number of repositories. Garlic will thus benefit from and contribute to work on query processing in heterogeneous distributed database systems as well as on distributed query processing in general (e.g., [17], [18]). Since Garlic queries are formulated in an object-extended SQL dialect, we also expect to build on work in the area of object query processing (e.g., [19], [20]). In addition to the problems raised by coupling heterogeneous databases and supporting object queries, Garlic also faces significant query processing challenges that arise in accommodating a broad range of repository types and data types.

Garlic will decompose a user's query into an execution plan containing a number of smaller queries, each of which can be executed by an underlying repository. It will be the wrapper's job to translate these smaller queries into the repository's native query language (or its native search API, if it has no actual query language). To do this decomposition, Garlic will need descriptions of the query processing power of each repository. The question of how to characterize the query power of a repository, in terms of the language subset that its wrapper is capable of processing directly (i.e., without help from the Garlic runtime system), is an interesting question that the project is currently investigating. Garlic will also need to come up with cost and selectivity estimates in order to devise an efficient access plan. Thus it will need cost- and selectivity-related metadata, which might be hand-written as part of the wrapper for repositories with fixed schemas, or might rely on query plan EXPLAIN facilities provided by more sophisticated repsoitories. For queries involving approximate-matching, new execution strategies are needed to produce the N best results efficiently (i.e., without materializing every intermediate result item that matches at all).

6: Garlic Interfaces and Applications

Data in Garlic will be accessible through two primary interfaces. Application programs will be able to access data in the system via Garlic's C++ application programming interface. Within this C++ API, two major sub-interfaces will be provided -- one for compiled applications, written with *a priori* knowledge of the Garlic schema for the database of interest, and one for dynamic applications, written with no *a priori* knowledge of the types or objects that exist in the database. End users will be able to access data in the system via Garlic's query/browser interface, a dynamic Garlic application (the first client of the C++ API) that will provide a friendly and intuitive means for users to query and browse the contents of any Garlic database. Unlike existing interfaces to databases, users will move back and forth seamlessly between querying and browsing activities, using queries to identify interesting subsets of the database, browsing the subset, querying the contents of a set-valued attribute of a particularly interesting object in the subset, and so on. The query/browser will support synchronous browsing [21], [22] and a graphically oriented extension of query by example [23]. An interesting question, currently under investigation, is how much of the power of the Garlic query language the query/browser will be able to make available without sacrificing ease of use and intuitive semantics. Finally, the query/browser will provide support for type-specific "pickers" to support the construction of media specific predicates.

7: Conclusions, Status and Future Work

We have presented an overview of the Garlic project at the IBM Almaden Research Center. Garlic's goal is to build a heterogeneous multimedia information system (MMIS) capable of integrating data from a broad range of data repositories. We described the overall architecture for the system, which is based on repositories, repository wrappers, and the use of an object-oriented data model and query language to provide a uniform view of the disparate data types and data sources that can contribute data to a Garlic database. A significant focus of the project is the provision of support for repositories that provide media-specific indexing and query capabilities.

An initial "proof of concept" prototype of Garlic should be running (or at least limping) by the time this proceedings becomes available. To speed development, we are using the ObjectStore DBMS [24] to hold both the complex objects and the metadata objects in the prototype. The initial prototype will be demonstrated via a simple application involving data that spans a relational DBMS (DB2 C/S), an image repository (based on QBIC), and a text repository. The initial query/browser prototype will be the front end for this application. The goal of the first prototype is to provide clearer insights regarding the nature of wrappers, the challenges involved in query translation and processing, and the efficacy of the query/browser as an end-user window into a collection of multimedia data.

In the longer term, we expect the Garlic project to lead us into new research in many dimensions, including object-oriented and middleware query processing technologies, extensibility for highly heterogeneous, data-intensive environments, database user interfaces and application development approaches, and integration of exact- and approximate-matching semantics for multimedia query languages. There are also many interesting, type-specific issues. such as what predicates should be supported on image and video data, how to index multimedia information, how to support similarity-based search and relevance feedback, and what the appropriate user interfaces are for querying particular media types. We believe that significant challenges exist in each of these areas, and that solutions must be found if the field is going to move beyond the traditional boundaries of database systems and keep pace with the emerging demand for large-scale multimedia data management.

8: Acknowledgments

We would like to thank Rakesh Agrawal for his input in the start-up phase of the Garlic project; he contributed significantly to our vision for both the project as a whole and the query/browser in particular. John McPherson and Ashok Chandra have been particularly supportive of our efforts throughout; we thank them for their encouragement and many suggestions. Many others contributed to the definition of the Garlic project, including: Kurt Shoens, K.C. Lee, Jerry Kiernan, Peter Yanker, Harpreet Sawhney, David Steele, Byron Dom, Denis Lee and Markus Tresch. We thank Markus Tresch and Yannis Papakonstanidou for their thoughtful comments on this paper.

9: References

[1] W. Niblack, et al., "The QBIC Project: Querying Images by Content Using Color, Texture and Shape", Proc. SPIE, San Jose, CA, February 1993

[2] A. Pentland, R. Pickard, and S. Scarloff, MIT Media Lab: "Photobook: Tools for Content Based Manipulation of Image Databases", Proc. SPIE, San Jose, CA, 1994.

[3] G. Salton, "Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer", Addison-Wesley Publishers, 1989.

[4] R. Rosenberg and T. Landers, "An Overview of MULTIBASE", in Distributed Databases, H. Schneider, ed., North-Holland Publishers, New York, NY, 1982.

[5] A. Elmagarmid and C. Pu, eds., Special Issue on Heterogeneous Databases, ACM Comp. Surveys 22(3), Sept. 1990.

[6] T. Conners, et al., "The Papyrus Integrated Data Server", Proc. 1st PDIS Conf., Miami Beach, FL, Dec. 1991.

[7] R. Ahmed, et al., "The Pegasus Heterogeneous Multidatabase System", IEEE Computer, 24:12, Dec. 1991.

[8] D. Fang, et al., "The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database Systems", Proc. ICDE, Vienna,, Apr. 1993.

[9] W. Grosky, "Multimedia Information Systems", IEEE Multimedia 1(1), Spring 1994.

[10] M. Ubell, "The Montage Extensible Datablade Architecture", Proc. ACM SIGMOD, Minneapolis, MN, May 1994.

[11] R. Cattell, ed., "The Object Database Standard: ODMG-93 (Release 1.1)", Morgan Kaufmann, San Francisco, CA, 1994.

[12] A. Black, et al., "Object Structure in the Emerald System", Proc. ACM OOPSLA , September 1986.

[13] F. Bancilhon, S. Cluet, and C. Delobel, "A Query Language for the O2 Object-Oriented Database System", Proc. DBPL Conference, Salishan Lodge, Oregon, June 1989.

[14] W. Kim, "A Model of Queries for Object-Oriented Databases", Proc. VLDB, Amsterdam, the Netherlands, Aug. 1989.

[15] K. Kulkarni, "Object-Oriented Extensions in SQL3: A Status Report", Proc. ACM SIGMOD, Minneapolis, MN, May 1994.

[16] H. J. Zimmermann, "Fuzzy Set Theory and its Applications", Kluwer Academic Publishers, Boston, MA, 1990.

[17] W. Du, R. Krishnamurthy, and M. Shan, "Query Optimization in Heterogeneous DBMS", Proc. VLDB Conference, Vancouver, Canada, 1992.

[18] C. Yu and C. Chang, "Distributed Query Processing", ACM Comp. Surveys, December 1984.

[19] S. Cluet and C. Delobel, "A General Framework for the Optimization of Object-Oriented Queries", Proc. ACM SIGMOD, San Diego, CA, June 1992.

[20] J. Blakely, W. McKenna, and G. Graefe, "Experiences Building the OODB Query Optimizer", Proc. ACM SIGMOD, Washington, DC, May 1993.

[21] A. Motro, A. D'Atri, and L. Tarantino, "The Design of KIVIEW: An Object-Oriented Browser", Proc. 2nd Int'l. Expert DB Systems Conf., Tysons Corner, VA, Apr. 1988.

[22] R. Agrawal, N. Gehani, And J. Srinivasan, "OdeView: The Graphical Interface to Ode", Proc. ACM SIGMOD, Atlantic City, NJ, May 1990.

[23] M. Zloof, "Query-By-Example: A Data Base Language", IBM Systems Journal 16(4), 1977.

[24] C. Lamb, et al., "The ObjectStore Database System", Comm. ACM 34(10), October 1991.

[25] O. Guenther, "Efficient Structures for Geometric Data Management", Lecture Notes in Computer Science, Springer-Verlag 337 (1988).

[26] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources", Proc. IEEE ICDE, Taipei, Taiwan, March 1995.

[27] G. Alonso, et al., "Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management", IBM Research Report, Sept. 1994.

Optimizing Queries Containing Path Expressions Using Alternative Logical Access Paths

Peter Schwarz[*] Laura Haas[†] Bart Niswonger[‡] Fatma Özcan[§]

IBM Almaden Research Center
San Jose, CA 95120

October 18, 1999

Abstract

Traditional optimizers assume that a query explicitly names the collection(s) in which the desired objects reside. In object-relational or object-oriented data models that support reference-valued attributes and path expressions, however, objects of interest may be reachable from several collections, or only indirectly via navigation from other objects. This suggests that optimizers for such data models could exploit alternative logical access paths to improve query performance.

This paper presents a comprehensive approach to the problem of multiple logical access paths. We present a simple language for specifying different paths that denote the same set of objects, and describe how such specifications can be used to discover interesting access paths. By introducing a CHOICE operator that can be inserted into a logical query plan, we show how to extend a traditionally-structured cost-based optimizer to evaluate multiple logical access paths, and we show how we obtain and use statistics to model the cost of particular choices. We present experimental results that demonstrate that the right choice of logical access path can improve performance by orders of magnitude, and show that our methodology for optimization and costing chooses wisely among the alternative plans.

1 Introduction

Traditional query optimizers rely on a few basic assumptions about how and where data is stored. In particular, the query explicitly names the (logical) collections that are to be searched, and it is assumed that no other collections can be used to access the same information. Increasingly, however, features are being added to database systems that weaken this assumption. For example, IBM DB2 Universal Database, an object-relational database that supports reference-valued columns and queries containing path expressions,

[*] *schwarz@almaden.ibm.com*

[†] *laura@almaden.ibm.com*

[‡] *bart@cs.washington.edu; current address: Department of Computer Science, University of Washington*

[§] *fatma@cs.umd.edu; current address: Department of Computer Science, University of Maryland, College Park*

requires the definition of each reference-valued column to specify a *scope*, or target collection, for references that will be stored there [CCN+97]. Other systems rely on the existence of *type extents*, collections that are known to contain all objects of the referenced type [KM90a, JWKL90].

Knowledge of a target collection gives the optimizer a choice of two paths by which to access the referenced data: by navigation from the path's source collection, or by direct access to the target collection. We call these options *logical access paths*, to distinguish them from indices or other *physical access paths* traditionally considered by optimizers. For each logical access path, there may be several physical access paths to choose from.

The preceding example describes a single instance of a more general phenomenon: database systems in which there can be multiple logical access paths for reaching the same data. The data models of object-oriented databases, such as ObjectStore [LLOW91], O_2 [LRV88] or Gemstone [BOS91], are even more flexible than those of object-relational systems. A given object may be reachable from many collections, or only indirectly via navigation from other objects. Object attributes can contain collections of references to objects, as well as scalar references.

If it is known that all of the objects of interest are reachable via a particular access path, it may be possible to choose an execution plan with much better performance. For example, consider the query:

```
select e.name, e.dept.name
from ResearchEmps e
where e.dept.size > 20 and e.dept.budget < e.dept.expenses
```

A naive execution plan would navigate from each employee in *ResearchEmps* to its department, and test the predicates. A more sophisticated optimizer would make use of the knowledge that the employee attribute *dept* is of type *Department*, and that all departments are included in a collection *Depts*. In addition to the naive navigational plan, this optimizer would be able to explore plans that, for example, take advantage of an index on *Depts* to find the large departments, and join the resulting rows with *ResearchEmps* to answer the query. However, a still more sophisticated optimizer would recognize that although the employee attribute *dept* is of type *Department*, employees in the *ResearchEmps* collection belong to research departments, which can be reached via the (much smaller) collection *ResearchDepts*. This optimizer could consider not only all of the previous plans, but also ones in which *ResearchEmps* is joined with *ResearchDepts*.

Federated systems, which integrate multiple data sources, exacerbate the problem of multiple logical access paths because independent collections of the same object type may exist at different sources. In the query above, the advantage of knowing that research employees reference research departments is even greater if the *Depts* collection contains references to departments stored by several data sources, making it expensive to scan, whereas the *ResearchDepts* collection refers to departments stored at only one location, perhaps the same one where the research employees are stored.

This paper presents a comprehensive approach to the problem of multiple logical access paths, developed in the context of the Garlic federated database system. Rather than requiring explicit scope declarations or relying on the existence of type extents, we define a specification language with which a database administrator can express a wide range of assertions concerning the objects reachable via a path. We present algorithms that analyze the assertions to discover useful logical access paths, and show how the ability to consider multiple logical access paths can be introduced into a traditionally-structured cost-based optimizer. Finally, we demonstrate experimentally the value of exploring alternative access paths, and show that our methodology for optimization and costing chooses a good execution plan.

Throughout, we stress pragmatism over completeness. We rely on assertions by an administrator rather than constraints, which are difficult to enforce in a federated system with autonomous data sources. We limit the expressive power of our specification language to simplify its application to queries. Rather than search exhaustively for every possible access path, our algorithms focus on finding those most likely to lead to good execution plans.

The paper is structured as follows. Section 2 describes our specification language, and Section 3 discusses the algorithms for discovering logical access paths. Section 4 concerns optimization. In Section 4.1 we describe how alternative access paths are represented in the query graph by means of the CHOICE logical operator, and we show how we obtain and use statistics to model the cost of particular choices in Section 4.2. Section 5 presents our experimental results, Section 6 discusses related work, and we summarize the paper in Section 7.

2 A Logical Access Path Specification Language

Although database metadata varies widely across data models and system implementations, conventional database schemas typically provide limited information from which to deduce alternative logical access paths. For example, Garlic is based on an object-oriented data model, and its schema includes information about the type of objects referenced from a collection, and the types of object attributes, method parameters, and the like. Thus, a Garlic schema might include an *Emps* collection, containing references to objects of type *Employee*. *Employee* objects might have an integer *Salary* attribute, as well as a *Dept* attribute of type *Ref<Department>*, and so forth. Note, however, that while the schema states the type of the *Dept* attribute explicitly, it includes no indication as to whether the referenced *Department* objects are included in (i.e., referenced from) a particular collection, e.g. *Depts*. Our specification language is a means by which to provide such information, in the form of simple inclusion assertions. For the purposes of this paper, we assume that the specifications are supplied by a knowledgable database administrator, and are correct.

For the example above, the logical access path specification might be:

$$\{Emps.Dept\} = F(Depts)$$

This specification states that all the distinct *Department* objects reachable via the path *Emps.Dept* can also be reached from the collection *Depts*.[1] The curly braces represent elimination of duplicates from a bag; thus the left-hand side represents distinct departments. The *F* on the right-hand side stands for *Filter*; if it were known that every department in *Depts* is referenced by at least one employee in *Emps*, a specification without a filter could be used instead:

$$\{Emps.Dept\} = Depts$$

If the target of a path spans multiple collections, a specification like the following could be used:

$$\{Emps.Dept\} = ResearchDepts, F(SalesDepts)$$

This states that the distinct departments reachable via the given path consist of all the departments referenced by the *ResearchDepts* collection, plus some departments referenced from the *SalesDepts* collection. More complex expressions built from these elements can describe a variety of interesting assertions. However, we do not claim that the language is capable of describing arbitrary relationships among collections. Our intent was to focus on assertions with practical implications for query processing, and to express such assertions with a limited number of constructs in a form easily understood by users or database administrators.

More generally, a *logical access path specification* has the form:

$$source = target$$

and states that the bags denoted by the expressions *source* and *target* are identical. The source expression must be a path, optionally enclosed by {} to indicate duplicate elimination. A path consists of a root (a named collection in the schema) followed by 0 or more steps.[2] Each step must be an attribute of the object type denoted by the previous step in the path. If any step in a path represents a collection-valued attribute, the collection is "flattened" before proceeding to the next path step. Thus, paths in specifications always denote bags of scalars, never bags of bags. For example, if objects of type *Department* have an attribute

[1] While the semantics of the specification language are those of bags, the examples in this paper assume that named collections (e.g. *Emps*) and collection-valued attributes (e.g. *Depts.emps*) do not contain duplicates.

[2] Thus, a simple collection can be represented as a path having a root and no steps.

Emps which contains the department's set of employees, the expression *Depts.Emps* denotes the bag union of all the employees in the departments referenced by the collection *Depts*, not a bag of bags of employees. While path expressions that are not necessarily flattened at each step are allowed in the query language, and analogous specifications are of theoretical interest, we believe our more limited form of specification is sufficient to describe most situations that will be encountered in practice.

The expression denoting the target of a specification is similar to the source expression, but may include the filter or union operators (written, respectively, as "*F()*" and ",") as well as the duplicate elimination operator. All of the operators take bag expressions as their argument(s), and produce a bag as their result. Restricting the source of a specification to a simple path, while permitting more complex expressions for the target, constrains the expressive power of the language but simplifies the application of specifications to a given query. The next section addresses the algorithms that implement this process.

3 Applying Logical Access Path Specifications to Queries

Before we describe how logical access path specifications are used, we will take a closer look at how the existence of an alternative logical access path can transform the query execution plan. Analogous to the distinction between logical and physical access paths, we distinguish a *logical plan* from an *execution plan*. In a logical plan, the collections to be queried are specified, along with any relevant predicates, but the variables considered by a traditional cost-based optimizer, (e.g join order and access method) are not. We focus on the logical plan here, while noting that several execution plans may exist for each logical plan. We will use SQL to express logical plans, because SQL provides exactly the right level of abstraction: the collections to be queried are specified, but there are no implications for join order, physical access method, etc. Whenever there is potential confusion as to whether an SQL statement represents a (user-submitted) query or a logical plan, we will be explicit.

We begin by noting that a path expression with a root and N steps can be viewed as an N-way join. For example, the query:

```
select e.name, e.dept.name
from ResearchEmps e
where ...
```

can be executed as a "pointer join", as represented by the following logical plan:[3]

```
select e.name, d.name
```

[3] Although the semantics of path expressions dictate that path steps through scalar attributes should be modeled as outer joins, our logical plan notation will use the more familiar and readable SQL for ordinary joins throughout.

```
from ResearchEmps e, e.dept d
where e.dept = d.OID and ...
```

This SQL statement represents the logical plan that locates qualifying objects of type *Department* by navigation from qualifying objects in *ResearchEmps*. The join predicate does not actually need to be evaluated in this case; it is guaranteed to be true because the *Department* objects are obtained by navigation. In the absence of alternative access paths, this navigational logical plan is the only feasible means of answering the query. However, if there is an alternative means of enumerating the objects over which the quantifier *d* ranges, it can be substituted for the definition of *d* to give an alternative logical plan. For example, if all the relevant departments are included in the collection *Depts* the logical plan becomes:

```
select e.name, d.name
from ResearchEmps e, Depts d
where e.dept = d.OID and ...
```

For this logical plan, the join predicate is important: it captures the idea that not every *Department* object in the *Depts* collection is necessarily reachable via the *dept* field of a research employee. Each object in *Depts* must be matched with an object in *ResearchEmps* that references it.

Given a query and a set of logical access path specifications, our algorithm systematically identifies substitutions like the one in the example above, and generates logical plans that take advantage of them. The algorithm proceeds in 2 phases. The *matching phase* determines, for each step in the path, collections or unions of collections that may be substituted for the corresponding quantifier definition. The *generation phase* produces logical plans by applying one or more of the substitutions identified during the matching phase.

3.1 Matching Phase

The matching phase builds a *substitution list* for each path step by starting with the path contained in the query, and looking for a logical access path specification whose source matches a prefix of that path. The matched prefix is replaced by the target of the specification, and the new path so formed is added to the substitution list. Both this new path and the original one are now eligible for further matching, and this process can be repeated as long as matching specifications are found.

For example, suppose the query contains the path *Emps.dept.mgr.name*, and the logical access path specifications given in Figure 1 have been registered. The substitution list for the first path step is initialized to the first step of the path in the query, *Emps.dept*, which matches the source of specification (1). Replacing the matched portion by the target of (1) gives *Depts*, so this collection is added to the substitution list for

$$\{Emps.dept\} = F(Depts) \tag{1}$$

$$\{Depts.mgr\} = F(Emps) \tag{2}$$

$$\{Depts.mgr\} = F(SalesMgrs, ResearchMgrs) \tag{3}$$

Figure 1: Logical Access Path Specifications

this step, after which no further substitutions are possible. Table 1 shows the completed substitution list for step one.

Choice	Collection
C1.1	Emps.dept
C1.2	Depts

Table 1: Substitution list for step one, *Emps.dept*

Choice	Collection
C2.1	Emps.dept.mgr
C2.2	Depts.mgr
C2.3	Emps
C2.4	SalesMgrs, ResearchMgrs

Table 2: Substitution list for step two, *Emps.dept.mgr*

For the second step, the list is initialized to *Emps.dept.mgr*. Applying specification (1) replaces *Emps.dept* by *Depts*, giving the new path *Depts.mgr*, which is added to the substitution list. This new path matches the source of specifications (2) and (3), so now each of these can be applied to add two more paths to the list: *Emps* and the union *SalesMgrs, ResearchMgrs*. At this point, no new matches are possible. Table 2 shows the completed substitution list for step two.

For the specification language described in Section 2, it is impossible to guarantee in general that the matching phase of the algorithm will terminate. In practice, this problem can be resolved in a variety of ways. One approach is to stop adding to the substitution list after some number of candidate substitutions have been found. Since the original navigational plan is available even if no substitutions are made, there is no danger of being unable to execute the query. Another option is to limit the maximum length of paths added to the substitution list, e.g. to twice the length of the path in the original query.

3.2 Generation Phase

At the end of the matching phase, a substitution list has been created for each step of the original path. Each entry in a step's substitution list identifies a collection (or union of collections) that contains all the objects denoted by that step of the path. The generation phase creates a logical plan by choosing one element from the substitution list for each path step, repeating this process until every combination of choices has been considered. However, the substitutions which can be applied at a given step depend on the substitutions that were made for preceding steps. A simple collection or union of simple collections can be substituted no matter what substitutions were made for preceding steps, but a collection represented by a path can be substituted only if each of the preceding steps chose the corresponding prefix of the same path. Thus, the number of logical plans actually generated is typically smaller than the total number of combinations. Note that this restriction is not strictly necessary, and eliminates from consideration any logical plans that have a greater number of joins than there are steps in the path expression being evaluated. We will return to this issue in Section 7.

```
select ...
from Emps e, C1 d, C2 m
where e.dept = d.OID and d.mgr = m.OID and ...
```

Plan	Choice 1	Collection C1	Choice 2	Collection C2
LP1	C1.1	*Emps.dept*	C2.1	*Emps.dept.mgr*
LP2	C1.1	*Emps.dept*	C2.3	*Emps*
LP3	C1.1	*Emps.dept*	C2.4	*SalesMgrs, ResearchMgrs*
LP4	C1.2	*Depts*	C2.2	*Depts.mgr*
LP5	C1.2	*Depts*	C2.3	*Emps*
LP6	C1.2	*Depts*	C2.4	*SalesMgrs, ResearchMgrs*

Table 3: Logical plans.

Referring to the choices in Table 1 and Table 2, if choice C1.1 is used for step one, choices C2.1, C2.3 and C2.4 are allowed in step two. C2.1 is allowed because C1.1 is its corresponding prefix. Conversely, C2.2 is disallowed because C1.1 is not its corresponding prefix. C2.3 and C2.4 are allowed because they are simple collections or unions of collections. Similarly, if choice C1.2 is used for step one, choices C2.2, C2.3, and C2.4 are allowed for step two. In total, six logical plans will be generated for this query, which are summarized in Table 3. The SQL above the table is a "template": logical plans LP1 - LP6 are constructed by substituting the collections given in the table for "C1" and "C2" in the template. The following examples illustrate the plans generated. For logical plan LP1, the complete SQL representation is:

```
select ...
from Emps e, e.dept d, d.mgr m
where e.dept = d.OID and d.mgr = m.OID and ...
```

This is the original navigational plan, with no substitutions. For LP5, the SQL representation becomes:

```
select ...
from Emps e, Depts d, Emps m
where e.dept = d.OID and d.mgr = m.OID and ...
```

In this case, the *Depts* and *Emps* collections have been substituted for the first and second steps of the path.

4 Exploiting Logical Access Paths For Query Optimization

The purpose of introducing logical access path specifications is to improve the performance of queries containing path expressions. We have described a language for specifying alternative paths, and an algorithm for enumerating alternatives relevant to a given query. In this section, we turn our attention to integrating this capability with traditional query optimization, so that we can generate execution plans based on these alternatives and select the best one.

Garlic processes queries following the Starburst model [HFLP89]. Queries are first parsed, and an internal, logical representation of the query is built in the form of a *query graph*. This representation is manipulated during Query Rewrite [HPH92], which heuristically transforms the query to arrive at a more "optimizable" form. Cost-based optimization [HKWY97, ROH99] then occurs, bottom-up, first deriving plans for single collection accesses, and then proceeding to build plans for joins from those.

4.1 The CHOICE Operator

To represent alternative logical plans, we added a new logical operator, CHOICE, to our query graphs [HP88]. As a query is parsed, Garlic uses schema information to look up any identifiers encountered, and builds pieces of query graph to represent them. When a path expression, P, is encountered, the logical access path specifications are consulted, and appropriate logical plans are identified using the algorithm in Section 3. Whenever there are multiple logical plans to select from, a CHOICE operator is introduced to represent P's output. For each of the alternative logical plans, a subgraph that completely describes the work required by that plan is built and hung below the CHOICE operator. All the subgraphs for P are semantically equivalent.

After query rewrite, the cost-based optimizer traverses the final query graph depth-first, optimizing each node of the graph (each logical operation) individually. The output of optimization is an *execution plan*, which consists of a tree of physical operators. To optimize a particular logical operation, optimized execution plans for its children in the query graph must already have been produced. The plan for the new logical operation typically builds on the plan(s) for its children by adding operators. For example, to optimize a logical join operation, execution plans for accessing the individual collections must already have been generated; the plan for the join usually consists of some flavor of physical JOIN operator combining the outputs of the execution plans for each child (or, for a multi-way join, perhaps a tree of physical JOIN operators). Likewise, to optimize the CHOICE operation, the optimizer finds the best execution plan for each of its various children (alternatives) first. Then it selects the best of those plans as the plan for the

CHOICE. CHOICE itself adds no additional operators to the execution plan.

Representing the logical access path alternatives explicitly in the query graph has a number of advantages. Since the query graph passes through both rewrite and cost-based optimization, rewrite can be used to expand the range of choices or to prune bad choices heuristically. The basic bottom-up nature of cost-based optimization is undisturbed, and existing code can be used to cost the individual alternatives. Because subgraphs of CHOICE can be arbitrarily complex, the query graph can express any alternative representable in the logical access path specification language – and more, since the CHOICE construct is available for other uses as well. For example, it could be used to represent the choice between using a materialized view or going straight to base data, or to represent the alternatives produced by chase/backchase in [DPT97].

A different approach to optimizing logical access paths would have delayed the exploration of alternative paths until cost-based optimization. In this approach, the path expression P would be treated as a single-collection access until the cost-based optimizer examined the possible access paths. The delayed approach has two potential advantages: it unifies logical and physical access path selection, and, because the cost-based optimizer can more easily re-use partial plans, it might allow certain efficiencies in optimization that will be harder to achieve using CHOICE. On the other hand, the delayed approach introduces a great deal of complexity into the cost-based optimizer, particularly because applying logical access path specifications can change the number of collections accessed to produce the query results. Most cost-based optimizers trying to plan a single-collection access are not well-prepared to optimize a join or union at that point.

The use of a logical operator to hold alternative rewritings of a query so that they could be optimized by the cost-based optimizer was suggested by Hasan and Pirahesh in [HP88]. In [GW89], Graefe and Ward introduce a *choose-plan* operator that can appear in execution plans. This operator allows the execution system to choose among physical plans based on the runtime value of program variables in the query – a similar idea, but applied in a different context, for a different purpose. We elected to use the logical operator, CHOICE, because it minimized the changes to our query compiler, preserving its basic structure while offering a great deal of flexibility for handling path expressions and other applications.

4.2 Selecting the Best Plan

The CHOICE operation gives us the ability to construct query graphs that represent alternative logical plans for evaluating path expressions. To select the best alternative, the optimizer must cost each one. Some of the alternatives may be simple accesses or joins, whose costs can be determined normally by the optimizer. In this section, our concern is with predicting the cost of those plans that involve navigation (i.e. pointer-chasing). In particular, we add to the framework of [ROH99] the ability to estimate two critical quantities in plans that employ pointer-chasing, and given these quantities, the ability to model the cost of navigation itself. The first quantity is result cardinality, i.e. the total number of objects denoted by the target of a path. The second quantity is predicate selectivity, which depends on the data value distribution of the attribute employed in the predicate. For example, consider the path expression *NYBranches.Depts.Emps.Age*. The result cardinality of this path expression is equal to the total number of employees in all departments of New York branches; the selectivity of a predicate containing this path depends on the value distribution of the *Age* attribute **for this particular group of employees.**

The emphasized phrase at the end of the previous paragraph highlights what makes the calculation of these quantities difficult in practice. The distribution of employee age at the New York branches may be quite different from the distribution of age values in groups of employees denoted by other paths, e.g. *CABranches.Depts.Emps.Age* or *USEmps.Age*. To collect and store such statistics comprehensively would require keeping track of them for every potential path, which is impractical at best and an impossibility in the common case of schemas containing cyclical relationships.

Nor is the problem of storing path statistics confined to value distributions. A path's result cardinality is also influenced by every step. The total number of objects designated by a path depends on the cardinality of the root collection and the *fanout* at each step (see, e.g., [BF93]). For example, the total number of employees designated by the first path above can be estimated based on the number of (New York) branches, the average number of departments in those branches, and the average number of employees in those departments. For the second path, *CABranches.Depts.Emps.Age*, any of these three numbers may be different. As in the case of value distribution statistics, keeping comprehensive fanout statistics for every potential path would be impractical or impossible.

Our approach to computing result cardinalities and predicate selectivities exploits the same logical access path specifications that are used to develop alternative logical plans. Rather than storing fanout and uniform value distribution statistics (second-highest key, second-lowest key and number of distinct values) for every possible path, we record them for attributes of simple collections only. In addition, we introduce a new statistic, **ref_collection**, which is derived using the logical access path specifications and maintained for reference-valued attributes of simple collections. This statistic identifies the simple collection or union of simple collections which provides the best available statistics for that attribute when it appears in a path expression. When more than one collection or union of collections could be used to approximate a particular path step, we use statistics from the collection (or union) with the smallest cardinality. Since all collections identified using the logical access path specifications contain a superset of the desired objects, we expect the smallest such collection to provide the best approximation.

We illustrate with an example. Consider the query

```
select e.name
from Emps e
where e.dept.name = 'Database'
```

If we have the single specification

$$\{Emps.Dept\} = F(Depts) \tag{4}$$

it seems clear that we would like to use the value distribution statistics relevant to *Depts.name* in predicting the selectivity of our predicate. But suppose *Emps* is a collection of research employees, and therefore we have also been provided with the specification:

$$\{Emps.Dept\} = F(ResearchDepts) \tag{5}$$

Assuming that the collection *ResearchDepts* is smaller that the collection *Depts*, we would intuitively expect that using the statistics associated with *ResearchDepts.name* would allow us to make more accurate predictions of the selectivity of the predicate.

Alternatively, suppose that instead of this last specification we are given the specification:

$$\{Emps.Dept\} = ResearchDepts, F(SalesDepts) \tag{6}$$

Perhaps the *Emps* collection collects together the technical employees of the company, some of whom are in sales. There are still probably fewer total departments in *ResearchDepts* and *SalesDepts* than in *Depts*, so we would prefer to use statistics based on those two collections. Using the statistics for either one alone, however, would clearly be wrong.

For the query in our example above, we would estimate the selectivity of the predicate by consulting the **ref_collection** statistic for the attribute *dept* of collection *Emps*. Depending on which logical access path specifications have been registered, we would be directed to different collections, as follows.

In the first case, where only specification (4) has been registered, the **ref_collection** statistic for the *dept* attribute of the *Emps* collection would point to the *Depts* collection. In the second case, in which specification (5) has also been registered, it would point to the *ResearchDepts* collection instead. In the third case, in which specification (6) has been registered instead of specification (5), it would point to both *ResearchDepts* and *SalesDepts*. The statistics for the union would be computed by combining the individual collection statistics, using the functions indicated in Table 4 for fanout, second-highest key and second-lowest key, and the rules given in Table 5 for the number of distinct values. The rules in Table 5 are used iteratively for unions involving more than two collections.

Statistic	Combining Function
high2key	Maximum
low2key	Minimum
fanout	Average

Table 4: Functions used to compute statistics from underlying, unioned collections' statistics

Condition	Rule for Computing **num_distinct**
$range_1$ is a subset of $range_2$	$num\_distinct = num\_distinct_2$
$range_2$ is a subset of $range_1$	$num\_distinct = num\_distinct_1$
$range_1$ overlaps $range_2$	$num\_distinct = max(num\_distinct_1, num\_distinct_2)$
$range_1$ does not intersect $range_2$	$num\_distinct = num\_distinct_1 + num\_distinct_2$

Table 5: Rule for computing the number of distinct values of an attribute from underlying, unioned collections' statistics. The range of an attribute spans the values between **low2key** and **high2key**. Rule is given for an attribute from two unioned collections; may be applied iteratively if more collections are being unioned.

Note that the **ref_collection** statistic for an attribute only directs the optimizer to a collection that provides the best available approximation to the desired statistics for paths with a single step. That is, the collection designated by the **ref_collection** statistic for *Depts.mgr* might not be the best possible approximation for the objects denoted by the path *Emps.dept.mgr*. A better approximation could be obtained by keeping **ref_collection** statistics for such longer paths as well. However, by traversing the **ref_collection** statistics along a path, it is possible to arrive at a reasonable approximation of the desired statistics without dedicating the processing power and storage that would be required to maintain more comprehensive statistics.

To clarify, we look at an example with a longer path expression. To calculate the selectivity of the predicate *Emps.dept.mgr.secretary.name = 'Smith'*, we need to find the value distribution statistics pertaining to the attribute *name*. Assume the following logical access path specifications have been registered:

$$\{Emps.Dept\} = F(Depts) \tag{7}$$

$$\{Depts.Mgr\} = F(Emps) \tag{8}$$

$$\{Emps.Secretary\} = F(Sctys) \tag{9}$$

$$\{Depts.Mgr.Secretary\} = F(ExecSctys) \tag{10}$$

The **ref_collection** statistic for *Emps.dept* tells us to find statistics for *mgr* by looking at the statistics for collection *Depts*. In turn, the **ref_collection** statistic for *Depts.mgr* tells us to find statistics for *secretary* by looking at the statistics for collection *Emps*, and, finally, the **ref_collection** statistic for *Emps.secretary* tells us to find the value distribution statistics for *name* by looking at the statistics for collection *Sctys*. Because we do not keep **ref_collection** statistics for longer paths, we fail to take advantage of specification (10) and thus do not look at the statistics of the *ExecSctys* collection, which should more closely approximate the objects targeted by our path. However, we believe that what we give up in potential accuracy is more than made up for by the savings in time and space.

In the preceding example, since the path included only scalar-valued attributes, the fanout at each step was one. However, if any of the attributes were collection-valued, we would use the **ref_collection** statistics to find the best available fanout statistics in exactly the same way.

The final step in costing a plan for evaluating a path expression is modeling the cost of the navigation itself. The cost of navigation is the cost of accessing the root of the path segment, fetching the attribute corresponding to the next step in the path, then fetching the attribute for the next step, and so on. Garlic keeps statistics that track the **avg_access_cost** for attributes [ROH99], that is, the cost, given an object identifier, to fetch a particular attribute for that object. Thus to compute the cost of navigating a path, we simply sum, for each step in the path, the cost of the necessary fetch to get the next attribute along the path, multiplied by the estimated cardinality of the path result thus far. This total is added to the cost to access the first collection of the path to arrive at a total cost for the plan.

5 Logical Access Paths in Action

In this section we demonstrate the power of using information about logical access paths to improve query optimization. In particular, we will show how the specification of alternative logical access paths increases the choices available to the optimizer, and that our optimizer uses this information wisely in determining a final execution plan. Along the way, we will show the utility of having union specifications, and indicate why merely having information about type extents does not always lead to choosing the best plan.

Our experiments were done using artificial data on a university theme[4]. The test data was distributed in a way that allows us to explore a large number of scenarios, but which is not in any way intended to be natural or realistic for an actual application. The collections, their sizes and their locations are shown in Table 6. A high-level view of the type system, showing references among the classes, is presented in Figure 2, and Table 7 contains the logical access path specifications for this database schema. Note that the specifications provide several ways of computing the result of a given path expression. For example, all objects reachable via the path *Courses.teacher* can be obtained both from the *People* collection and the *Professors* collection.

[4] We actually borrowed the data from the BUCKY benchmark [CDN+97], but modified the schema and distributed it to serve our experimental needs.

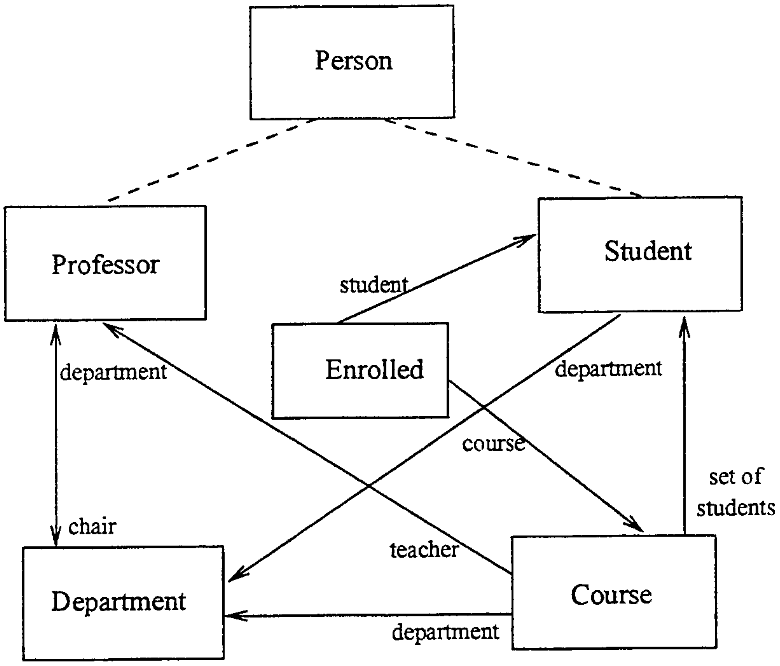


Figure 2: Inter-Class References

The test data is distributed among three data sources: an IBM DB2 Universal Database (UDB) relational database, a Lotus Notes version 4.5 database, and an ObjectStore version 4.0 object database. The wrappers [RS97] for these sources (and indeed, the sources themselves) have very different capabilities. The relational wrapper exposes most of the capabilities of its powerful underlying source. It can apply any predicate, do most joins (with the exception of OID to OID joins), and accept binding values for predicates. The Notes database is much less powerful; it does no joins, and the wrapper does not accept binding values for predicates. However, it does accept and execute most predicates that do not contain unbound variables. Most primitive of all is our ObjectStore wrapper, which provides only the minimal capabilities required of all wrappers, namely, the ability to select all OIDs from a given collection, and the ability to retrieve any attribute, given an OID. It does no projection or selection (or joins, of course).[5]

We run our experiments in a distributed environment as well. Notes runs on an NT server; UDB and ObjectStore run on one RS/6000 server, while Garlic itself runs on another. All servers are on a LAN, with normal daily workloads. Each query was run multiple times, and we report an average time value.

[5] ObjectStore itself, of course, is capable of doing all of these operations.

Collection Name	# of Objects	Location	Type (Referenced)
Enrolled	100,000	UDB	Enrolled
Courses	1250	ObjectStore	Course
Departments	250	ObjectStore	Department
Physical_Sciences	84	UDB	Department
Social_Sciences	83	ObjectStore	Department
Humanities	83	Notes	Department
Students	25,000	ObjectStore	Person
Professors	25,000	ObjectStore	Person
People	50,000	UDB	Person

Table 6: Collections of the Example Schema

{People.department} = F(Physical_Sciences, Social_Sciences, Humanities)
{Professors.department} = F(Physical_Sciences, Social_Sciences, Humanities)
{Students.department} = F(Physical_Sciences, Social_Sciences, Humanities)
{Courses.department} = F(Physical_Sciences, Social_Sciences, Humanities)
{Courses.teacher} = F(People)
{Courses.teacher} = F(Professors)
{Courses.students} = F(People)
{Courses.students} = F(Students)
{Enrolled.student} = F(People)
{Enrolled.student} = F(Students)
{Enrolled.course} = F(Courses)
Physical_Sciences.chair = F(People)
Physical_Sciences.chair = F(Professors)
{Physical_Sciences.chair} = F(Professors)
Social_Sciences.chair = F(People)
Social_Sciences.chair = F(Professors)
{Social_Sciences.chair} = F(Professors)
Humanities.chair = F(People)
Humanities.chair = F(Professors)
{Humanities.chair} = F(Professors)
{People.department} = F(Professors.department, Students.department)
{Professors.department} = F(People.department)
{Students.department} = F(People.department)

Table 7: Logical Access Path Specifications for the Example Schema

5.1 A First Example

In our little "university," each area manages its own information about its departments. The humanities area, for example, stores its information in the collection *Humanities* in Notes, while the social sciences and physical sciences store theirs in ObjectStore and UDB, respectively. There is a collection of information on university people (students and professors) in UDB. A user desiring information about people in a particular department might issue a query such as:

```
select p.age
from People p
where p.department.name = 'deptname99'
```

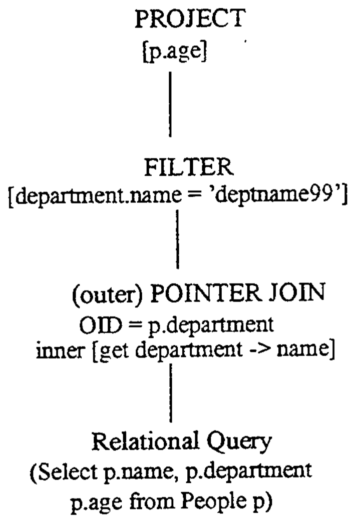


Figure 3: Execution Plan for *People* Query Without Logical Access Paths

If logical access path specifications are not used, the optimizer has no choice but to use the plan shown in Figure 3: a straight path traversal, scanning *People*, and for each person, fetching the relevant department information (indicated as "Pointer Join" in the plan, this operation actually consists of a method invocation to get the values of the desired attributes), then testing whether it matches the predicate. As there are 50,000 people in the database, this is a rather slow procedure.

Using the logical access path specifications, however, the optimizer has considerably more freedom. The plan it chooses, shown in Figure 4, scans the three collections of departments, retrieving the department name. It then filters out all but the relevant department, and uses its OID to probe UDB for the people in that department. As there are only 250 departments, this plan is considerably faster – in fact, almost 200 times faster, as shown in Table 8. (The predicate on department name could be pushed down through the union, further enhancing performance – a rewrite rule would accomplish this).

Note that our schema includes a collection, *Departments*, that contains references to all the departments and thus could serve as a type extent. One might wonder whether a plan that scanned *Departments*, and then did the join with *People* wouldn't be faster than the one chosen by the optimizer. (The optimizer did not, in fact, evaluate the suggested plan, since no logical access path specification {*People.department*} =

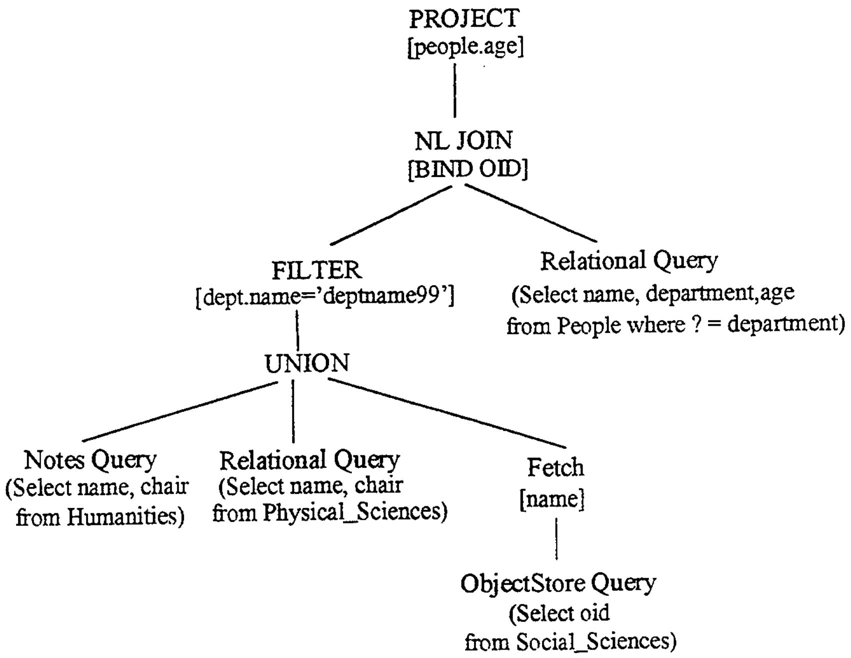


Figure 4: Execution Plan for *People* Query With Logical Access Paths

	Without Log. Acc. Paths	With Log. Acc. Paths
Compilation Time	.121 secs	.268 secs
Execution Time	1312.91 secs	7.022 secs

Table 8: Compile and Execution Time of the *People* Query

F(Departments) was registered). However, note that the *Departments* collection stores only references to the actual departmental data. Therefore, for each department reference obtained, it would still be necessary to access one of the three sources to fetch its name – causing a lot of back and forth traffic between Garlic and the data sources. It is unlikely that this would be a better plan than the union, which very efficiently gathers the department information. Hence, the existence of a type extent does not automatically indicate a preferred path; depending on where the data are located, and on the query itself, the type extent may or may not be an efficient means of access to the data.

5.2 Another Example

The *Enrollments* collection in our schema is a relationship table linking students to the classes they are taking. All the information about university people (students and professors) can be found in the *People* collection; however, there is a collection in ObjectStore, *Students*, containing references to just the students.

To find information about a particular student's department and courses, a user could issue the following query:

```
select e.student.department.name, e.course
from Enrolled e
where e.student.name = 'studentName97245'
```

Figure 5 shows the query execution plan produced by the optimizer when it is given no logical access path specifications. As in the previous example, there is no choice – the only way to execute the query is by navigation from *Enrolled*, selecting the OID of the student, then using method invocation to fetch first the student information, and then, whenever the enrollment record refers to the target student, the information about his or her department. As *Enrolled* is a very large table, and method invocation is an expensive operation, this takes about 2.4 hours to execute.

When provided with logical access path specifications, however, the optimizer has many more choices (note the increase in optimization time in Table 9). It chooses the plan shown in Figure 6. This plan is much more efficient, executing in under five seconds, as shown in Table 9.[6] Instead of starting with *Enrolled*, this plan goes to *People*, applying the predicate there, and using the OID of the single student that results to probe the *Enrolled* table, from which just a handful of records (two, to be precise) are returned. Since the number of records is so small, the cost to fetch the matching department information via navigation is insignificant, so a Pointer Join is used. Note that the values of *Enrolled.student* can be found in either *Students* or *People*, yet the optimizer chose to use *People*. Again, this is a sensible decision: iterating through the ObjectStore collection *Students* would have required fetching *name* and *department* from UDB for each of the 25,000 students before the predicate could be evaluated, whereas by using *People* the predicate can be applied to the data directly.

	Without Log. Acc. Paths	With Log. Acc. Paths
Compilation Time	.174 secs	.837 secs
Execution Time	8682.71 secs	4.69 secs

Table 9: Compile and Execution Time of the *Enrollment* Query

5.3 Discussion

In this section, we have shown how knowledge about logical access paths can speed up the execution of queries by several orders of magnitude. We have also shown that the implementation we described works, and is able to make good decisions among logical access plans.

[6] An even better plan would have been to push the entire join down to the relational wrapper. This was not feasible because our relational wrapper does not currently support this particular kind of join on OID.

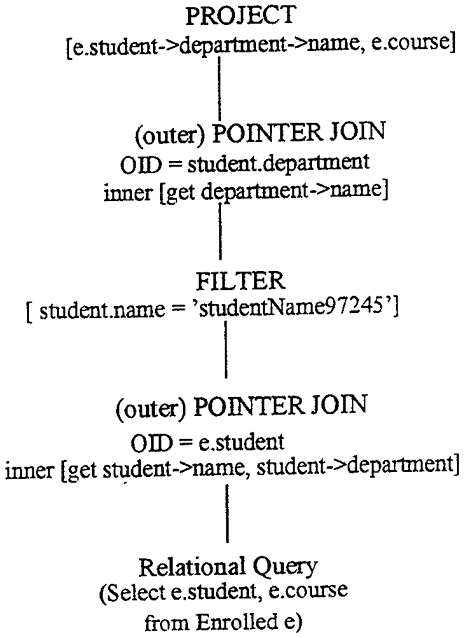

PROJECT
[e.student->department->name, e.course]
(outer) POINTER JOIN
OID = student.department
inner [get department->name]
FILTER
[student.name = 'studentName97245']
(outer) POINTER JOIN
OID = e.student
inner [get student->name, student->department]
Relational Query
(Select e.student, e.course
from Enrolled e)

Figure 5: Execution Plan for *Enrollment* Query Without Logical Access Paths

Of course, there is no gain without some pain. In introducing alternative logical access plans, the search space of feasible execution plans is increased; hence, the compilation time of a query also increases. In Tables 8 and 9, we see the compilation time of the example queries with and without the use of logical access path specifications. Although the compilation time increases considerably (by a factor of five, for example, in Table 9), for these and for many queries the amount of time saved in execution more than justifies the extra time spent in compilation. In general, the compilation time grows linearly in the number of alternative logical access plans. Hence for queries with several complex path expressions, in an environment in which there are many logical access path specifications, compilation time can become significant (especially if the data sets are small, or local, or for other reasons, cheaply accessed). Heuristic rules could be devised to control whether to use the logical access plan mechanism in these situations.

The schema and configuration we used for our tests was admittedly artificial, being designed for testing purposes, and not for a real application. However, the kinds of logical access path specifications we used, will, we believe, prove to be common and important, especially in middleware that integrates legacy systems. It is not uncommon for there to be duplicate collections of information in such environments, any one of which can answer a query. Likewise, partitioning of information, exemplified in our schema by the departmental data, arises frequently in any organization with subdivisions. We expect that other applications of these kinds of specifications will be prompted by the evolving need to store XML documents relationally. Logical access path specifications could provide support for untyped or variable-typed references, needed

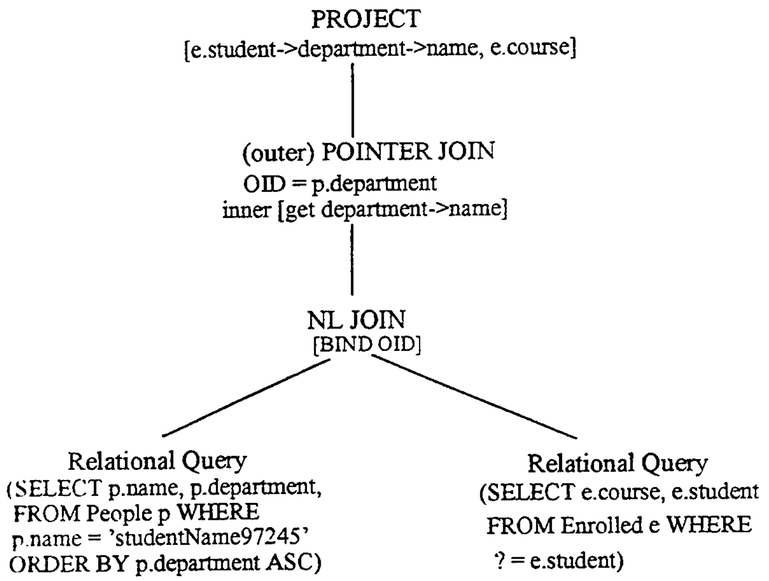

PROJECT
[e.student->department->name, e.course]
(outer) POINTER JOIN
OID = p.department
inner [get department->name]
NL JOIN
[BIND OID]
Relational Query
(SELECT p.name, p.department,
FROM People p WHERE
p.name = 'studentName97245'
ORDER BY p.department ASC)
Relational Query
(SELECT e.course, e.student
FROM Enrolled e WHERE
? = e.student)

Figure 6: Execution Plan for *Enrollment* Query With Logical Access Paths

for efficient queries through XML IDREFs when the XML is decomposed into several relational tables for storage [STH+ 97].

6 Related Work

There have been many papers on the optimization of queries with path expressions. In this section we try to indicate the most relevant; we do not attempt a comprehensive survey!

In [SC90], Shekita and Carey demonstrated conclusively the value of having techniques other than simple pointer-chasing for executing path expressions. Since then, there have been a number of proposals for using type information (i.e., class extents) to transform pointer-chasing into more optimizable joins [KM90a, JWKL90, LV93, CD92, BMG93]. [KM90a, JWKL90] focus on the minimization of I/O by using class extents to transform pointer chasing into algebraic join operations. [LV93, CD92] present frameworks which combine type-based query rewriting with standard algebraic optimization. [BMG93] proposes a complete optimization framework, which distinguishes between the logical algebra and the execution algorithms. They introduce a new logical operator, *materialize*, which allows them to consider segments of a path as independent nodes, and provide a corresponding set of logical transformation and implementation rules. Our work builds on these papers; in addition to type and schema information, we allow a database administrator to provide additional specifications which our system can also use to convert path expressions into explicit joins.

Cost models for object-oriented operations are proposed in [GGT95, GGT96, AOD96, BF93, MW99]. These papers by and large provide more detailed models than ours, as they take into account indexes, clustering, and other physical characteristics that are not relevant to our system (because Garlic does not store its data). They track similar attribute statistics to ours. [GGT95, GGT96] take into account clustering, indexes and embedded objects in a detailed physical model. [BF93] proposes a set of parameters to model the topologies of object references in an object-oriented database supporting inheritance hierarchies. [AOD96] employ formulas from statistics and probability theory to estimate the selectivity of path expressions and the size of joins. [MW99] introduces an optimization framework for XML queries. They track statistics for paths of length less than or equal to k. Our cost model is generally not as sophisticated as those presented here, as that is not the focus of our work (and because we do not need to worry about the physical characteristics of the data storage [ROH99]). However, we present a novel mechanism for inferring the required statistics for paths of arbitrary length using basic statistics for paths of length one and the logical access path specifications. Our estimates could be improved by using some of the more detailed statistics presented in these papers.

Another class of related papers uses constraints and other local information such as views to help find efficient query plans [Lev99, DPT97, AV97]. [Lev99] surveys the use of views in answering queries. [AV97] makes use of constraints in answering queries, but focuses heavily on validating the constraints; we leave this task to the definer of the logical access path specifications. [DPT97] uses constraints to define both logical and physical access paths, and proposes a general mechanism for generating alternative logical plans using *chase* and *backchase*. Our logical access path specifications focus more narrowly on logical alternatives for path expressions. However, we handle cases, such as types with no extent or types with an extent formed by a union, that require some extension to the framework of [DPT97]. We also expect our more focused algorithm for generating alternative logical plans to be more efficient.

Other papers in this general area include works on secondary index structures such as path indices [BW89, Ber91] and access support relations [KM90a, KM90b]. Such techniques are more pertinent to the design of object-storage systems than to middleware environments where such structures are expensive to build and maintain. However, if indexes are available they can be exploited in our framework as part of cost-based optimization; this issue is orthogonal to our work. [CCM96, MW99] propose query optimization techniques for generalized path expressions (primitives that allow data and structure to be queried uniformly), enabling search over semi-structured data. While we use our logical access path specifications in an environment with a fixed database schema, semi-structured database systems might also benefit from using such information in planning execution of generalized path expressions.

7 Summary and Conclusions

Data that can be accessed via multiple logical access paths is rapidly becoming a reality in database systems that support today's more flexible data models. This paper has shown how a traditional cost-based optimizer can be adapted to explore alternative logical access paths, and that doing so leads to better execution plans for queries.

The first part of the paper described our logical access path specification language, the means by which a database administrator makes the system aware of alternative paths. The language is quite expressive, but deliberately limited in scope so that specifications are easier to write and easier for the system to exploit. Further research and experience is needed to determine if our language strikes the right balance between expressive power and usability.

Obviously, the optimizer will only make correct decisions if the specifications provided by administrators are correct. Care must be taken, therefore, to register only those assertions that reflect persistent structural properties of the organization or system being modeled. We believe that such properties are plentiful in real applications. Nevertheless, mechanisms for enforcing or at least verifying assertions expressed in the specification language are another important area for future work. Finally, the greater the number of specifications provided, the more choices the optimizer will be able to explore in its search for the best plan. This suggests that mining tools capable of examining the data and suggesting possible specifications to the database administrator could be very valuable.

The second part of the paper described our algorithm for enumerating alternative logical plans, given a query and a set of access path specifications. Our algorithm does not find all the alternative logical plans; in particular, plans in which extra joins are added to the query are not considered. While this may seem counter-intuitive, there are cases where adding a join can lead to a better execution plan, and further research is needed to determine whether such opportunities are sufficiently frequent to warrant the increase in complexity and optimization time needed to recognize them.

The next portion of the paper discussed the implementation of our approach within a system based on a traditional cost-based optimizer. They fit together well: by introducing a CHOICE logical operation into the query graph, alternative logical plans can be evaluated without disturbing the basic bottom-up nature of cost-based optimization. Furthermore, because the CHOICE operation and all the alternatives are represented in the query graph, they are available during the query rewrite phase of optimization as well. Rewrite rules that expand the range of choices available to the optimizer or heuristically prune unpromising choices are another fertile area for additional study.

The final part of the paper demonstrated experimentally that our optimizer can take advantage of logical access path specifications to find plans with dramatically better performance than could be achieved otherwise. However, exploring a larger space of plans drives up optimization time, and may not be warranted when the potential payoff is small. Further experiments are needed to better understand the tradeoff between increased optimization cost and decreased execution cost.

Our approach to optimizing the evaluation of path expressions extends and generalizes previous work in which schema information is used to convert path expressions into joins. Its flexibility and pragmatic nature should prove particularly valuable in the context of federated database systems, for which the benefit in recognizing alternative logical access paths can be especially significant, but for which techniques that rely on tighter control over the data are difficult to implement.

References

[AOD96] M. Altinel, C. Ozkan, and A. Dogac. A cost model for path expresions in oql. In *Proc. of the European Joint Conf. on Engineering Systems Design and Analysis*, Montpellier, France, June 1996.

[AV97] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 122–133, Tucson, AZ, USA, May 1997.

[Ber91] E. Bertino. An indexing technique for object-oriented databases. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 160–170. IEEE Computer Society, 1991.

[BF93] E. Bertino and P. Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 1993.

[BMG93] J. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, DC, USA, May 1993.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The Gemstone object database management system. *Communications of the ACM*, 34(10):64–77, 1991.

[BW89] E. Bertino and K. Won. Indexing techniques for queries on nested objects. *tse*, 1(2):196–214, June 1989.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[CCN+97] M. Carey, D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, and N. Mattos. O-O, what have they done to db2. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 542–553, Edinburgh, Scotland, August 1997.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Diego, CA, June 1992.

[CDN+97] M. Carey, D. DeWitt, J. Naughton, M. Asgarian, P. Brown, J. Gehrke, and D. Shah. The BUCKY object-relational benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–146, Tucson, AZ, USA, May 1997.

[DPT97] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints and optimization with universal plans. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, August 1997.

[GGT95] G. Gardarin, J-R Gruser, and Z-H Tang. A cost model for clustered object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Zürich, Switzerland, September 1995.

[GGT96] G. Gardarin, J-R Gruser, and Z-H Tang. Cost-based selection of path expression processing algorithms for object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, September 1996.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–366, Portland, OR, USA, May 1989.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HP88] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst, August 1988. IBM RJ 6367.

[HPH92] J. Hellerstein, H. Pirahesh, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, San Diego, USA, June 1992.

[JWKL90] B. P. Jenq, D. Woelk, W. Kim, and W.-L. Lee. Query processing in distributed ORION. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the International Conference on Extending Database Technology : Advances in Database Technology*, volume 416 of *LNCS*, pages 169–187, Berlin, March 1990. Springer.

[KM90a] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Atlantic City, NJ, May 1990.

[KM90b] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, 1990.

[Lev99] A. Levy. Answering queries using views: A survey. submitted for publication, 1999.

[LLOW91] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The ObjectStore system. *Communications of the ACM*, 34(10):50–63, 1991.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 424–433, Chicago, IL, USA, May 1988.

[LV93] R. S. G. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 363–373, Dublin, Ireland, 1993.

[MW99] J. McHugh and J. Widom. Query optimization for xml. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, UK, September 1999.

[ROH99] M. Tork Roth, F. Ozcan, and L. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, UK, September 1999.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[SC90] E. Shekita and M. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Atlantic City, NJ, May 1990.

[STH+97] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 302–314, Edinburgh, Scotland, August 1997.

The Rufus System: Information Organization for Semi-Structured Data

K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, J. Thomas
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

While database systems provide good function for writing applications on structured data, computer system users are inundated with a flood of semi-structured information, such as documents, electronic mail, programs, and images. Today, this information is typically stored in filesystems that provide limited support for organizing, searching, and operating upon this data. Current database systems are inappropriate for semi-structured information because they require that the data be translated to their data model, breaking all current applications that use the data. Although research in database systems has concentrated on extending them to handle more varieties of fully structured data, database systems provide important function that could help users of semi-structured information.

The Rufus system attacks the problems of semi-structured data. It provides searching, organizing, and browsing for the semi-structured information commonly stored in computer systems. Rufus models information with an extensible object-oriented class hierarchy and provides automatic classification of user data within that hierarchy. Query access is provided to help users search for needed information. Various ways of structuring user information are provided to help users browse. Methods associated with Rufus classes encapsulate actions that users can take on the data. These capabilities are packaged in a framework for use by applications. We have built two demonstration applications using this framework: a generic search and browse application called *xrufus* and an extension to the Usenet news reading program *trn*. These applications are in daily use at our research laboratory.

This paper describes the design and implementation of our framework, our experiences using it, and their influence on the next version of Rufus.

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

1 INTRODUCTION

The volume and diversity of the information stored on computer systems have grown with the systems themselves. Current workstation users store gigabytes of information locally and have access to far more over local area networks. While some of this information is highly structured and stored in databases, most of it is stored in ordinary files arranged in a directory tree.

It is difficult for people to make effective use of the information that's available to them. The large amount of data makes it difficult to find things when they are needed, while the diversity of information makes it difficult to use the data when it is found. Since computer systems offer little help locating and using data, users are compelled to memorize the location of data and procedures for using it. The tools provided by current computer systems are crude and do not scale as needed.

For example, consider the plight of an organization with hundreds of internetworked workstations. Users of these workstations write documents using any of a dozen word processing systems. Although the workstations make use of shared file systems like NFS [17], users must still locate documents of interest by filename. The system might offer a brute force application for searching through files, but the applications tend to be slow and to make it difficult to find what a user needs. Once a document is found, the user needs to remember how to browse or print the document using the application specific for its type. While this example uses word processor documents, the same situation holds for computer programs, images, electronic mail, configuration files, and so on.

Ideally, computer systems would provide significantly better tools for users to manage huge amounts of data. This ideal system would know what each piece of data is and how to use it. In the document example above, the system would know what application(s) apply to each file and how to run them. The ideal system would also know what's inside each piece of data to allow users to search for information about a particular subject. The searching should adapt to the data type, with different techniques available for searching for text, images, and coded data. The ideal system would use indexing so that queries could be answered quickly. Finally, the ideal system would know the relationships between various pieces of data. For example, in a document that includes figures, the system should understand the inclusion relationship.

In contrast to the ideal system, today's users must choose between storing their data in traditional filesystems or in database systems. For various reasons, filesystems have little or no semantics attached to stored files. An attempt to add these semantics to an existing system would likely break all existing applications and creating a new system from scratch, with all new applications, is unthinkably expensive.

Alternatively, users could store their data in a database. Unfortunately, database systems are unprepared to store the semi-structured information inundating users. Instead, database systems are oriented towards providing high integrity storage for structured data. Database research has concentrated on supporting the same type of data more efficiently, with better concurrency, and with better integrity. Efforts to extend the scope of data that database systems can handle have succeeding in capturing more applications with fully structured data, but still do not support semi-structured data.

There are two reasons why current database systems are inadequate for storing semi-structured data. The biggest inhibitor is that database systems insist on "owning" the data. When you decide to use a database system, you convert your data into its format and access the data exclusively through the database system. Moving semi-structured information into a database abandons all the applications that were written against the data's original format.

Another problem is that semi-structured data is imperfect—computer programs may have syntax errors or be incomplete, documents may not format correctly, and electronic mail may be damaged by the delivery system. A database solution designed to store this information must be able to represent imperfections. Database systems are instead oriented towards storing perfect information and for providing facilities for keeping it perfect. This need to cope with imperfection motivates filesystems to maintain unintrusive byte-stream models.

In summary, given the choice between byte-stream filesystems and structured databases, users have chosen filesystems for storing their semi-structured data. This is an unfortunate choice, because database systems offer many features that could help users cope with information overload. Database systems need to step up to the problems of semi-structured data to make these features available.

The Rufus project brings features traditionally belonging to database systems to bear on semi-structured information. An object-oriented database is used to store descriptive information about file system objects. To preserve existing applications, Rufus does not modify the file system objects themselves. An import process automatically categorizes each piece of user data into one of the Rufus classes and creates an object instance to represent the data. The underlying database supports fast querying and object access. Rufus provides various ways of structuring the objects to support browsing. This object infrastructure is made available through a client-server interface. We have built two applications to demonstrate the value of our infrastructure.

While designing the Rufus system, four particularly interesting problems were addressed. The first problem is the automatic classification of a file into one of the Rufus classes. Such a classifier must be fast, accurate, and easily extended with new classes. The second problem is correlating file system objects with Rufus objects. The most obvious approach, using file names, fails when files are renamed but users expect object identity to be maintained for the new file name. The third problem is the ability to add and delete classes from the class hierarchy of an existing database. With traditional approaches, such schema changes break the class hierarchy, due to the class relationships established by inheritance. The fourth problem is that of maintaining a dynamically-extendible text index with concurrent readers. This problem is further complicated by the need to be able to scale text indices to hundreds of thousands or millions of objects.

This paper describes the design and implementation of the Rufus system. Particular attention is focused on how the system addresses the four key problems listed above.

2 RELATED WORK

While existing data management systems do not support semi-structured information, research in the areas of object-oriented database systems, information retrieval, classification, hypertext, and some specific applications contribute useful techniques.

Semantic file systems [11] (SFS) provide *transducers* that extract attributes from files and provide them to an indexing system. Queries against a semantic file system are issued via extensions to the file naming syntax of the UNIX file system and are presently limited to conjunctive equality tests with string prefix matching. Rufus and SFS share the goal of raising the level of abstraction of the file system interface. Embedding the query language in the file naming mechanism provides access to SFS facilities without changing applications, but can be unnatural for some queries. As currently described, SFS does not associate actions with data, nor does it represent inter-file relationships. Users need richer data modeling and query capabilities to cope with the millions of files available to them.

Intelligent mail filtering capabilities, such as those found in BBN/Slate [5], the Andrew Message System [25], the Information Lens [18] and Tapestry [13] give users a large measure of control over their incoming mail. The term "mail" is used rather loosely here, as these systems purposely blur the distinction between traditional point-to-point electronic mail and point-to-many bulletin-board message systems. Tapestry, in particular, advocates replacing the notion of sender directed mail by recipient directed content-based retrieval. Thus the receiver, rather than the sender, controls what the receiver sees. Tapestry takes an active approach by providing user-defined intelligent agents to forage through various mail/message databases for items of interest. Collaborative retrieval is supported in Tapestry by associating user annotations with messages. These annotations can be used by others to select messages to read.

In many ways Tapestry is an information retrieval system applied to a limited interactive mail/message domain. In contrast, full-text information retrieval systems such as RUBRIC [19] and WAIS [15] provide users with the ability to retrieve files as the result of queries posed against the document text. To facilitate retrieval across a wide variety of document formats, these systems treat their data as unformatted text. Information retrieval systems and specialized systems such as mail handlers can be thought of as at opposite ends of the "domain specificity" spectrum. Rufus supports both kinds of use. A general purpose application can provide access across all data types, while special purpose applications can be written to exploit the semantics of specific data types. We describe both kinds of applications later in this paper.

Object-oriented database systems, such as ObjectStore [21] and O_2 [8], provide explicit frameworks for describing the structure of data types. These systems provide powerful query languages that allow users to express retrievals based on the structures defined in the schema. Object-oriented systems also provide a simple framework for encapsulating an object's structure together with its semantics, or behavior. As with other database systems, use of an OODBMS requires that a user's data reside in the database.

Database systems do not really concern themselves with modeling and importing existing file types and files. Mechanisms are usually provided for the one-time import of users' files, but the expectation is that they will then "live" in the database world: applications that were used to manipulate the original data are not applicable to the proprietary, internal database formats. Additionally, little or no support is provided for refreshing the imported data from native files that may have changed (through the use of external applications). Users either step fully into the database world or are left to manage the consistency of the two worlds on their own.

Hypermedia systems [6], which are based on a browsing metaphor rather than one of retrieval, also use proprietary internal data formats. Systems such as Intermedia [29] provide no avenues for integrating existing structured data into a hypermedia document other than as flat text. Once a hypermedia document or web is created, the systems offer only limited access paths to the underlying data. For example, although Intermedia is built on top of a relational database, the relational query capabilities are not available to Intermedia users. Hypermedia systems require that the data they operate upon be brought into the system, as is the case with database systems. Most of the value is derived from careful construction of links, which must be added by hand. Finally, hypermedia systems do not encode information about how to operate on data once it is found.

3 RUFUS

This section describes the Rufus approach to supporting semi-structured information. We describe how each aspect of the design is implemented in our current prototype, our experiences with the design choices, and modifications we are making to Rufus based on these experiences. To avoid confusion, "Rufus" refers to our general approach, "Rufus 1" refers to our current prototype, and "Rufus 2" refers to the version that we are in the process of building based on our experiences.

Rufus augments the file system representation of user data with persistent objects that retain information extracted from the original data. The original data is not modified and remains the authoritative copy so that existing applications are not affected. Rufus provides a set of classes that describe types of user files; examples include mail messages, C language source code, and various image file formats. Rufus includes a classifier that automatically determines the Rufus class of a file.

The structured objects that Rufus extracts are used for querying, organizing, and operating upon the data. In addition to extracting structured information about user data, Rufus indexes the data's contents. Rufus 1 supports a full-text index for textual data; other types of indexing could be added for non-textual data. Queries on Rufus data combine search operators on the contents of data (full-text in Rufus 1) with predicates on the extracted objects. Rufus uses collections, sub-classing, composite objects, and hypertext links to represent structure between objects. Some examples: 1) a mail folder is modeled as a collection of mail message objects; 2) a C program is modeled as a composite object including collections of source and object code, compilation instructions, and documentation; and 3) the structure of questions and answers in bulletin board articles is represented with hypertext links.

Rufus 1 is implemented on a client/server model to mimic the location transparency provided by distributed file systems. Rufus applications are written using a client library that provides program access to queries and Rufus data. Rufus 1 includes a catalog server to locate active Rufus servers. We have written two applications, one general purpose and one data-specific, as Rufus clients. These applications are in daily use at our laboratory.

Figure 1 shows the general structure of the Rufus system.

3.1 Classifier

The classifier examines a file and guesses what its Rufus class is, providing the first piece of information that a user needs about a piece of data. Given the volumes of semi-structured data, it is unreasonable to expect people to classify information manually. Thus, automatic classification is needed. Successful classification permits Rufus to dispatch the correct import method to extract attributes from user data.

The classifier uses the presence of keywords, file name patterns and file type (directory or normal file), and the presence of constant bit patterns near the beginning of the file ("magic numbers"). For efficiency, the classifier scans the file once to prepare an abstract of sampled keywords from the beginning, middle, and end of the file. The keyword samples include the token to the left of the keyword to pick up punctuation in examples like `\section` in LaTeX.

Each class supplies an evaluation function that returns a weight from 0 to 10 according to how likely the data is a member of the class. For sufficiently nondescript data, the classifier will likely return TEXT (plain ordinary text) for files that have mostly printable characters or BINARY for anything else. In case of error, the user may manually reclassify an object.

For the set of classes we have defined, the classifier is reasonably accurate and fast. To test it, we classified 847 examples of various file types. 90% were classified correctly, 8% were editor backup files that are given unusual names and were classified as "Text" instead of their actual type (mostly C language source code), and 2% were more significant errors (most were telephone directories classified as "Text"). A similar test on 100 randomly-selected user files revealed 4 significant misclassifications. Two were text

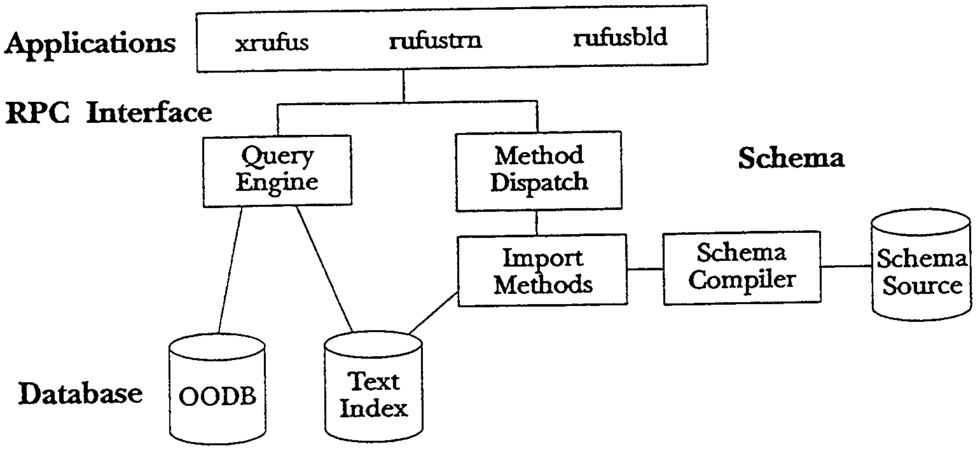


Figure 1: Structure of Rufus System

formatter documents generalized to "Text," one was an extremely short text editor command script misclassified as "Binary," and the last was a command script misclassified as a specific kind of script. On these tests, the classifier averaged 55 milliseconds of CPU time per file on an IBM RISC System/6000, model 350.

The larger problem is that the classifier must carefully balance the weights returned by the evaluation functions to make the right decision in most cases. It would be difficult or impossible to add many more classes without upsetting this balance.

To address these limitations, we are building a new classifier based on a different model. In this new classifier, the programmer describes salient features of a new type, such as binary numbers that should appear near the beginning or regular expressions that should be found in textual formats and provides examples of objects of the given type. A classifier training program collects all the unique features into a global *feature vector* and computes the centroids of the feature vector for each type for which samples are available.

The actual classification of an object is performed by constructing its feature vector and matching it to the class with the nearest centroid. We are currently using the cosine coefficient similarity measure, a dot-product of two feature vectors normalized to remove biases towards classes with many features.

To date, we have tried this new classifier technique on a set of about 45 types, including those that Rufus 1 supports. The results are encouraging: the new classifier is about as accurate as the previous version and the process of adding new types is simple to it is simple. We now need to improve the performance of the new classifier (it can take it several seconds to classify a file) by combining the regular expression features into a single finite automaton with an algorithm like that suggested in [1] and to build the feature vectors for each class in a single pass through the file.

3.2 Importing Data

Once a piece of user data has been classified as class C, it can be imported into Rufus. Importing simply means that attributes are extracted from the underlying data and stored as an object. If the underlying data is not perfectly formatted, values may be left out of the extracted object. If the class of the object is text-oriented, the textual contents of the data are added to the text index.

Each Rufus class provides an import method that's responsible for performing extraction. Although the writers of classes are free to choose any formalism they wish to analyze the underlying data, Rufus neither supplies nor dictates the use of any such formalism. We have used plain C code for extraction in the classes we have implemented so far.

When a file is imported into Rufus for the first time, a new object identifier is created for it. If the file is modified and re-imported, its object identity is retained. When files are renamed, it can be difficult to maintain object identity. To cope with this problem, Rufus uses a type-specific unique identifier to track file identity, rather than the file name. For example, mail messages have unique "message identifiers" associated with them. Plain Unix files can be identified by their "inode" and "device" numbers.

When a file is imported, its unique identifier is discovered by its class's constructor. A persistent mapping from unique identifier to object identifier is consulted; if the unique identifier is already known to Rufus, then the existing object identifier is reused. Rufus converts the varying-length unique identifiers to fixed-size object identifiers for convenience.

The Rufus strategy for unique identifiers works well for data that has an intrinsic unique identifier. For cases where Rufus must rely on the Unix file identification, our scheme works as long as file identity and object identity remain in sync.

Rufus includes a utility for importing data called *rufusbld*. Rufusbld reads a user-written specification that describes the files to be imported, classifies the files, and imports them into Rufus. Rufusbld does little work for files that are already "current" in the Rufus database, so an affordable way to keep Rufus current is to periodically run rufusbld. We decided against a strategy of hooking Rufus into the operating system's file system interface to avoid non-portable, system-dependent programming.

We tested rufusbld on a sample of 1,000 USENET articles. On our IBM RISC/System 6000 model 350, it takes about 130 milliseconds of CPU time per article imported, exclusive of classification. The real time to import is about 2.5 times the CPU time, due to waiting for database disk I/O's.

3.3 Data Model

The Rufus data model represents structured information observed about user data. The structured view describes what the data is, interesting values determined about the data, and what operations can be performed on it. We chose an object-oriented (OO) model [12]. The classes in the hierarchy are recognizable to users as types of information that they use. Rufus creates an object to represent each piece of information that a user might think of as distinct. For example, Rufus creates an object for each file of C source code, as well as an object for the `makefile` (compilation instructions) and an encompassing object for the entire program that refers to the constituent source code, `makefile`, documentation, etc.

A Rufus class is defined by a set of *attributes* associated with each instance of the class and a set of *methods* that can be applied to any instance of the class. Rufus supports substitutability, where instances of a class can be used in places that expect instances of a superclass. This capability allows users to take a specific or general view of a piece of data. For instance, one might seek a document that contains a particular phrase, without regard to the type of formatter used to compose it.

Attributes of type *context* define parts of the underlying real data for text indexing, similar to location restrictions in information retrieval systems like STAIRS [14]. For example, a document class might define the context *abstract* to refer to the words in the up-front summary of a paper. Contexts allow users to restrict the locations that words or phrases must appear in so that more accurate results are possible. For convenience, a class may indicate that particular string-valued attributes are to be indexed as contexts.

Every Rufus class includes a set of standard attributes and methods. The standard attributes include the object identifier, the document identifier (used to cope with new versions of the object in the text index), a unique identifier derived from the underlying object for correlation, the object's class, the date the object was last refreshed, and a string description of the object for browsing. Standard methods *display* an object, *import* an object into Rufus, and *print* an object. The standard methods provide the set of basic services that any Rufus object is expected to support.

Although methods defined for a Rufus class may choose to modify the underlying data, Rufus provides no built-in mechanism for mapping modifications to Rufus objects to the underlying data. Such a mapping would be difficult to provide, given that Rufus objects do not typically model all the information in the underlying data.

Our prototype currently has 34 classes, including a few formats of electronic mail, several formats of documents, C language source code, a few image types, bibtex citations [16], and employee records from IBM telephone books. Figure 2 shows a subset of the supported classes.

We chose the object-oriented (OO) model because its features closely follow a user's mental model of data. For example, OO data models feature a strong notion of object identity, while other data models are oriented around values. Object identity gives us an easy way to refer to objects in different contexts; in particular, it allows us to organize the same objects in different ways. Object identity is also useful for modeling complex objects made up of simpler ones. Rufus uses the attributes of objects to describe features extracted from the underlying data (e.g., author, title, and date written). Rufus uses the methods of an ob-

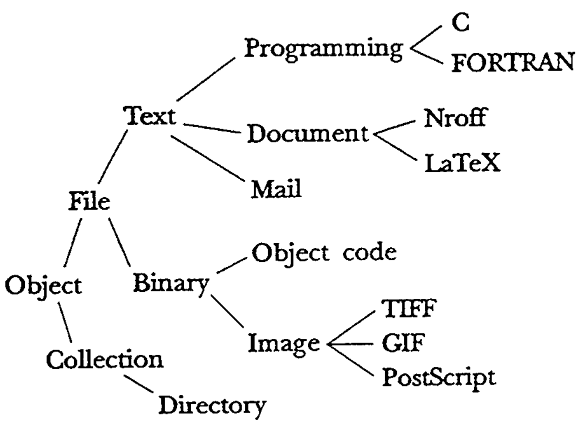


Figure 2: Subset of Rufus class hierarchy

ject to describe both user-visible operations that can be performed, as well as operations needed by the Rufus infrastructure, such as *display* and *import*.

Effective exploitation of the class system requires high quality, detailed class definitions. For example, Rufus 1 extracts the sender, newsgroup, subject, line count, summary, and organization fields from USENET articles, as well as creating links that show the "question and answer" relationship between articles. In contrast, the C source code class does not provide the same level of detail: it only distinguishes between *string* and *comment* contexts and does not distinguish between function definitions and references. As a result, Rufus 1 is more helpful for manipulating USENET articles than for C source.

While the object-oriented model has served our purposes well, some things are difficult to describe in a single-inheritance hierarchy. For example, in Rufus, embedded PostScript is a sub-class of IMAGE, which in turn is a subclass of BINARY. PostScript is textual rather than binary, though. In addition, while most people probably think of embedded PostScript as an image format, others see it as a programming language, requiring different treatment.

Schema changes in our prototype are discouraged because they invalidate existing databases. We note that schema evolution is a common problem in object-oriented database systems.

We are adopting a significantly different OO data model to address the above problems. Our new system uses the conformity data model of Emerald [3] and Melampus [22]. In this data model, only the methods of an object are visible outside its class definition. For convenience, an attribute can be marked so a method will be generated to return its value. In the conformity model, the suitability of an object for a purpose is dictated by whether it implements the necessary method names with the right parameter lists. For example, a user might be looking for objects that have an *Author* and a *Title*. In the system, objects of type LaTeX, TROFF -MM, and SGML might conform to this specification by implementing these two methods.

In the conformity model, inheritance is decoupled from subtyping. Schema evolution is simplified by the resulting independence between class definitions. Inheritance is not used to structure the class hierarchy, but rather as a modularity and reuse aid. When a class definition changes, the old definition of the class will be retained as long as there are object instances of the old class. All new object instances will use the new class definition.

In the new data model, class definitions are machine independent, so that client applications can retrieve class definitions from servers to interpret objects, even on different architecture hosts.

We have a working class compiler for the new data model and modifications to Rufus that fetch and store the new types of objects, keep track of the types of collections, dispatch methods on objects, and execute simple queries. We have so far implemented a few types in the conformity model, including FILE, TEXT, and RFC822 (mail messages).

3.4 Example of a Class Definition

This section presents an example of a Rufus class definition. The definition of the RFC822 class (electronic mail as passed over the Internet) is used. RFC822 is a subclass of MAIL. The table below lists some of the attributes extracted from RFC822 format methods:

Attribute	Data Type	Meaning
length	integer	Length of message
filename	string	File message stored in
messageid	string	Unique identifier
posted	date	Date written
subject	string	Subject of message
to	string list	List of recipients
from	string	Message sender

In addition, RFC822 supports contexts that contain the header fields of the message, the "Subject:" field of the message, the sender of the message, and the body of the message.

The RFC822 class supports several methods, among them:

Method	Meaning
display	Format message for display
edit	Edit the file containing the message
reply	Compose a reply to the message
forward	Forward message to someone else

The "reply" and "forward" methods bring up parts of a pre-existing application to perform these tasks. Users like to locate a message with Rufus, then apply their usual mail-reading tools to it.

In the original Rufus classifier, RFC822 format messages were recognized by the existence of "Received" and "To" fields in the header of the message. In the new classifier, RFC822 format messages are required to have a line beginning with "Received:" or "Delivery-Date:"; other features indicative of the format are lines beginning with "From:" "To:" and "Message-Id." We currently use about 30 samples of RFC822 format messages to train the classifier.

3.5 Structuring Concepts

While it is important for users to be able to understand facts about an individual object, it is also important to understand the relationships between objects. By making these relationships explicit, users are freed from having to know or discover them. Structure information is particularly helpful to applications that support browsing.

Rufus provides collections, object composition, subclassing, and hypertext links [6] to represent inter-object structure. Collections are sets of objects. An object can be in several collections at once. Collections themselves are objects and can be stored in collections as well. For each class, Rufus maintains a collection of all instances of the class, called the *class extent*. Collections are also used to store the results of queries. The objects in a collection can be from arbitrary classes.

Rufus uses object composition to represent complex things made up of other objects. For example, a C program is modeled as the composition of a `makefile`, C source code, and documentation. Rufus complex objects are represented by allowing the use of object identifiers as object attributes.

Subclassing is used in Rufus to indicate the specialization of object types. For example, an implementation might model filesystem directories as a subclass of collections. A filesystem directory does everything that a collection does, in addition to which it has file system attributes like filename, owner, modification date, etc.

Finally, hypertext links model system-discovered and user-specified connections between otherwise unrelated objects. For example, entries in the traditional Unix manual refer to other entries. The import method for Unix manual entries can create links to represent the cross-references. Likewise, a user writing a textual annotation of an image might establish a link to represent the relationship. In Rufus, links are separate objects that point to the linked objects. Fine granularity of link endpoints is achieved using type-specific *selection identifiers*, which are fixed length bit strings that an object's class can convert into a specific part of the underlying native data. In contrast with traditional hypermedia systems, Rufus does not modify the original data to represent links.

3.6 Query Language

The Rufus query language extends content-based access to semi-structured data. Rufus queries combine predicates on the objects extracted from the underlying data with predicates on the underlying data content. Rufus 1 supports simple object predicates and text search.

A Rufus 1 query searches a collection or class extent and returns objects that match a predicate. The predicate contains boolean combinations of conditions on the objects' attributes and text search predicates on the underlying real data. Attribute conditions are simply relational tests against constants, such as `posted > date(12/10/92)`.

More powerful query capabilities would be useful. For example, suppose one were looking for a message written during an electronic mail conversation with a colleague. In order to find the set of messages that comprise the conversation, one would like the query language to be able to follow the links established by the "In-Reply-To" fields of the messages and compute the transitive closure. We chose a simple subset to implement for expediency and to capture the most immediate needs. We are considering a new query language based on the set-oriented operators of the Melampus query language [23] to address these needs.

For text predicates, we implemented *near* (words close to each other) and *adjacent* (words close to each other in the right order). The boolean combinations supported by the query language provide the usual *and*, *or*, and *not*. A special optimization is made to execute text predicates like "*phrase1* `and not` *phrase2*" efficiently. The proximity and boolean operators can be nested to pose queries like `near(adj(San Francisco) earthquake)`.

Rufus supports stemming [27] and flexible capitalization. Stemming is based on a dictionary of 10,000 root forms and allowable stems. Users can state that they wish to ignore capitalization, want an exact match, or want at least the first letter capitalized.

Boolean and proximity text search have been thoroughly criticized [4]. We selected them as our initial text search capability because the results are easily explained to users. We are adding approximate searching based on term weighting [24] and relevance feedback [26] ("find me more documents like *these*") to Rufus.

3.7 Text Indexing

Rufus maintains an index on the text content of imported files to support fast searching on their content. We implemented a text index due to prevalence of textual data. For flexibility, we selected inverted files with word locations. Inverted files support both traditional boolean and proximity searching [27] and term-weighted searching [24]. We used fixed-size small blocks to represent the inverted file so that it can be updated incrementally. For our intended applications, we find that most of the objects indexed are unchanged from day to day, so incremental indexing makes refreshing the Rufus database significantly faster than a complete rebuild. The inverted file can also be updated concurrently with query access. A B-tree is used to store the starting block number of the inverted list for each indexed word.

For the sake of experience, we support two large Rufus databases. One covers a week's worth of Usenet articles (about 35,000); the other covers many weeks of IBM internal bulletin board articles (about 130,000). We found that the text index dominated the size of the Rufus data, the time to refresh the Rufus databases, and query performance. We were able to realize significant improvements with some simple modifications. Further improvement is expected when we change our stored structures.

When we measured the Rufus text index, we found that a huge number of words were being indexed. As a test, we selected 20,000 random articles from the Usenet article base. When indexed, they yielded more than 400,000 unique words and more than 8,000,000 word occurrences. We developed a stop list of the 280 most common words that eliminates 44% of the word occurrences. Random sampling of the vocabulary revealed that many time-stamp based identifiers and meaningless words derived from textual encoding of binary data were being indexed. Refinement of the constructor for the USENET class to avoid indexing such material reduced the vocabulary by one half. This example illustrates the advantage of specializing import according to the data's class.

We took a small random sample of the remaining vocabulary and classified each word by hand. Here's what we found:

Category	%
Proper names	20%
Real words	18%
Machine names, userid's	18%
Program symbols	13%
Misspellings	8%
Addresses, ZIP codes, phone numbers	7%
Other junk	5%
Acronyms	4%
Message-ID's	4%
Codes	3%

In our text database, we also found that the storage scheme of using fixed size blocks suffered from internal fragmentation and poor locality. Due to the distribution of word frequencies, many words have few occurrences. These infrequent words take up an entire small block, wasting space. Other words are more common and take up many small blocks, requiring many seeks to resolve a query. We are replacing our fixed block text inverted list implementation with a variable length block scheme such as that described in [9]. Briefly, this scheme uses small initial blocks, then scales up to larger blocks as the list of occurrences for a word grows. Efficient storage is achieved for infrequent words, while longer word lists are clustered better, reducing seeks.

While textual data is prevalent, indices oriented towards other data types would be useful. For example, the QBIC (Query by Image Content) project [20] at IBM Almaden is working on searching medical images. Their algorithms could be provided in Rufus for image data in addition to the text search capabilities already supported.

3.8 Client/Server

Since users are provided access to much of their data through distributed file systems, the Rufus capabilities described in the foregoing are implemented by servers to provide the same connectivity to data. Each Rufus server mediates access to a single Rufus database. Access to Rufus server functions is provided through a client-callable library of routines. These routines provide the ability to connect to Rufus servers; create, destroy, and modify objects; iterate through collections; invoke methods; and pose queries. In turn, the client library routines invoke functions in the Rufus server using an RPC mechanism.

Rufus servers provide their own concurrency control to allow queries and data import to run in parallel without threatening the physical integrity of the Rufus database. The concurrency control that Rufus wields does not apply to the underlying files. Due to the asynchronous updating of the Rufus database with respect to the underlying data, it is possible to locate files via queries that should no longer match the query predicate We have considered additional processing to drop query results that should not match due to changes that occurred since import but have not done so The larger problem is locating query results that now *should* be included in the answer but do not due to a stale Rufus database; for that we have no answer without modifying the operating system.

In Rufus 1, client applications can connect to a single Rufus server at a time. This limitation puts the burden on the user to figure out the correct Rufus server to use. We are removing this restriction in Rufus 2 so that clients will be able to connect to several servers at once Objects in one database will be able to point to objects in other databases. The new system will also be able to access servers supporting other remote protocols, such as WAIS [15] with Z39.50 [2]. Conversely, our system will export a Z39.50 protocol itself, so that its databases can also be searched by WAIS clients. Servers will be able to swap class definitions between themselves, using the same mechanism as is used to inform clients of class definitions.

Servers will be able to publish a summary of the information they store to permit automatic routing of queries to only those servers that might have useful information.

4 Applications

We wrote two applications to demonstrate the Rufus capabilities. Xrufus, an X-windows [28] application, provides querying, browsing, and operation execution over any of the data types known to Rufus. An object found by querying or browsing is displayed in xrufus according to the object's class. In addition, a menu of operations is prepared specific to the type of object. For example, the menu for electronic mail objects contains actions like "reply" and "forward."

Users can create *buttons* that represent commonly useful queries. For example, a user might define a "Callup" button that looks up a name in the site telephone book. Then, the user can select a name with the mouse in most X-windows applications, and click the button to run the query.

With more class definitions and integration, xrufus could be extended into the "researcher's workbench." Activities like processing mail, reading bulletin boards, program development, document processing, appointment scheduling, and talk preparation could all be provided in a seamless environment. The Rufus infrastructure would help users find and organize their information and to drop into the right applications at each step without having to think about them.

We've also developed an extension of the popular *trn* news reading program [7] called *rufustrn*. Rufustrn works just like trn, in addition to which users can define *virtual newsgroups* that contain all the articles that match a Rufus query. For example, a user might select specific articles from a newsgroup based on content to cut down the number of articles that must be examined. Alternatively, a subject of interest might appear in several newsgroups. This subject can be collected into a single virtual newsgroup for convenience. Rufustrn also allows users to pose queries of "one time" interest and browse the results. A nice feature of rufustrn is that articles are always displayed with the standard trn user interface. The result is a news reader enhanced with query capabilities, rather than a completely new application. The approach of rufustrn differs somewhat from that used in Infoscope [10]. Infoscope defines virtual newsgroups in a DAG structure based on the contents of other virtual newsgroups and of header fields. In contrast, rufustrn defines virtual newsgroups as the result of a query. Rufustrn provides a single mechanism for both one-time queries and for topics of continuing interest.

We envision supporting further applications beyond rufustrn. For example, a mixed database of text and multimedia data could allow users to search for film clips by searching through textual descriptions and invoking methods to view related clips.

To support such an application, Rufus only needs to have a "film clip" data type added with a method that invokes a video viewer on the user's workstation.

5 CONCLUSIONS

Users are inundated with semi-structured information. Current database systems do not handle such information well. As a result, users are forced to turn to specialized applications that improve access to particular kinds of data. Each specialized application is forced to re-invent and re-implement basic infrastructure to support flexible access. For structured information, database systems provide standard capabilities that make applications easier to write. The same leverage must now be applied to semi-structured information.

The Rufus project has developed an infrastructure based on object-oriented database and text search principles to support applications using semi-structured information. Applications built with the Rufus infrastructure remember key information that users would otherwise be forced to memorize, such as the relationship between files, how to find them, and what to do with them when you find them. Rufus raises the level of abstraction so that users no longer have to deal with their data as simple sequences of characters. We have built a prototype to demonstrate the Rufus ideas and deployed it for use at Almaden. Early experience with the prototype has been promising and has suggested important areas for further work. While we built our prototype on a UNIX system, we expect the Rufus concepts to be useful in other operating system environments as well.

The extensions being made in our new Rufus prototype will support applications on a significantly larger scale. Improvements in the storage structures will support databases with millions of objects. The work in distributed access will free users from specifying where to search for information and will integrate users' environment with information available in external information servers and libraries. The new conformity data model will allow new classes to be written and refined to support new kinds of data.

Acknowledgements Eli Messinger wrote both versions of the Rufus classifier. This paper benefited from the helpful feedback given by the reviewers.

References

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), 1975.

[2] ANSI/NISO, New Brunswick, NJ. *Information Retrieval Service and Protocol*, 1989.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of ACM Conference on Object Oriented Programming Systems, Languages and Applications*, September 1986.

[4] D. Blair and M. Maron. An evaluation of retrieval effectiveness for a full-text retrieval system. *Communications of the ACM*, 28(3), March 1985.

[5] Bolt, Beranek, and Newman, Inc., Cambridge, MA. *BBN/Slate Topics Manual*, 1990.

[6] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9), 1987.

[7] W. Davison. *TRN—Threaded Read News Program*, 1992.

[8] O. Deux et al. The O2 system. *Communications of the ACM*, 34(10), October 1991.

[9] C. Faloutsos and H. Jagadish. On B-tree indices for skewed distributions. In *VLDB Conference*, 1992.

[10] G. Fischer and C. Stevens. Information access in complex, poorly structured information spaces. In *Proceedings 1991 CHI Conference*, 1991.

[11] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic file systems. In *SOSP91*, October 1991.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, 1983.

[13] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(2), 1992.

[14] IBM. *STAIRS General Information Manual.*

[15] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. Technical Report TMC-199, Thinking Machines Corporation, Cambridge, MA, 1991.

[16] L. Lamport. LaTeX*: A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[17] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), February 1990.

[18] T. Malone, K. Grant, F. Turbak, S. Brobst, and M. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5), May 1987.

[19] B. McCune, R. Tong, J. Dean, and D. Shapiro. RUBRIC: A system for rule-based information retrieval. *IEEE Transactions on Software Engineering*, 11(9), September 1985.

[20] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content using color, texture, and shape. *SPIE 1993 International Symposium on Electronic Imaging: Science & Technology, Conference 1908, Storage and Retrieval for Image and Video Databases*, February 1993.

[21] Object Design, Inc., Burlington, MA. *ObjectStore User Guide*, 1991.

[22] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proceedings of ACM SIGMOD Conference*, 1991.

[23] J. Richardson and P. Schwarz. MDM: An object-oriented data model. In *Proceedings of the Third International Workshop on Database Programming Languages*, August 1991. Also available as IBM Research Report RJ 8228, San Jose, CA, July 1991.

[24] S. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3), 1976.

[25] J. Rosenberg, C. Everhart, and N. Borenstein. An overview of the Andrew Message System. In *Proceedings of SIGCOMM '87 Workshop*, August 1987.

[26] G. Salton. *Relevance Feedback and the Optimization of Retrieval Effectiveness*, chapter 15. Prentice Hall, 1971.

[27] G. Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.* Addison-Wesley, 1989.

[28] R. Scheiffler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[29] N. Yankelovich, B. Haan, N. Meyrowitz, and S. Drucker. Intermedia: The concept and construction of a seamless information environment. *IEEE Computer*, 21(1), January 1988.

Type Classification of Semi-Structured Documents*

Markus Tresch, Neal Palmer, Allen Luniewski

IBM Almaden Research Center

Abstract

Semi-structured documents (e.g. journal articles, electronic mail, television programs, mail order catalogs, ...) are often not explicitly typed; the only available type information is the implicit structure. An explicit type, however, is needed in order to apply object-oriented technology, like type-specific methods.

In this paper, we present an experimental vector space classifier for determining the type of semi-structured documents. Our goal was to design a high-performance classifier in terms of accuracy (recall and precision), speed, and extensibility.

Keywords: file classification, semi-structured data, object, text, and image databases.

1 Introduction

Novel networked information services [ODL93], for example the World-Wide Web, offer a huge diversity of information: journal articles, electronic mail, C source code, bug reports, television listings, mail order catalogs, etc. Most of this information is semi-structured. In some cases, the schema of semi-structured information is only partially defined. In other cases, it has a highly variable structure. And in yet other cases, semi-structured information has a well-defined, but unknown schema [Sch93]. For instance, RFC822 e-mail follows rules on how the header must be constructed, but the mail body itself is not further restricted.

The ability to deal with semi-structured information is of emerging importance for object-oriented database management systems, storing not only normalized data but also text or images. However, semi-structured documents are often not explicitly typed objects. Instead they are, for instance, data stored as files in a file system like UNIX. This makes it difficult for database management systems to work with semi-structured data, because they usually assume that objects have an explicit type.

Hence, the first step towards automatic processing of semi-structured data is *classification*, i.e., to assign an explicit type to them [Sal89]. A classifier explores the implicit structure of such a document and assigns it to one of a set of predefined types (e.g. document categories, file classes, ...) [GRW84, Hoc94]. This type can then be used to apply object-oriented techniques. For example, type-specific methods can be triggered to extract values of (normalized and non-normalized) attributes in order to store them in specialized databases such as an object-oriented database for complex structured data, a text database for natural language text, and an image database for pictures.

Such a classifier plays a key role in the Rufus system [SLS$^+$93], where the explicit type of a file, assigned by the classifier, is used to trigger type-dependent extraction algorithms. Based on this extraction, Rufus supports structured queries that can combine queries against the extracted attributes as well as the text of the original files. In general, a file classifier is necessary for any application that operates on a variety of file types and seeks to take advantage of the (possibly hidden) document structure.

Classifying semi-structured documents is a challenging task [GRW84, Sal89]. In contrast to fully structured data, their schema is only partially known and the assignment of a type is often not clear. But as opposed to completely unstructured information, the analysis of documents can be guided by partially

*Research partially supported by Wright Laboratories, Wright Patterson AFB, under Grant Number F33615-93-C-1337.

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

available schema information and must not fully rely on a semantic document analysis [Sal89, Hoc94]. As a consequence, much better performing classifiers can be achieved. However, neither the database nor the information retrieval community have come up with comprehensive solutions to this issue.

In this paper, we present a high performance classifier for semi-structured data that is based on the vector space model [SWY75]. Based on our experiences with the Rufus system, we define high performance as:

- *Accuracy.* The classifier must have an extremely low error rate. This is the primary goal of any classifier. For example, to be reliable for file classification an accuracy of more than 95% must be achieved.
- *Speed.* The classifier must be very fast. For example, to be able to classify files fed by an information network, classification time must be no more than 1/10 of a second.
- *Extensibility.* The classifier must easily be extensible with new user-defined classes. This is important to react to changing user demands. The ability of quick and incremental retraining is crucial in this context.

The remainder of the paper is organized as follows. In Section 2, we review the basic technology of our experimental vector space classifier and compare the accuracy of several implementation alternatives. In Section 3, we introduce a novel confidence measure that gives important feedback about how certain the classifier is after classifying a document. In Section 4, we show that finding a good schema is crucial for the classifier's performance. We present techniques for selecting distinguishing features. In Section 5, we compare the vector space classifier with other known classifier technologies, and conclude in Section 6 with a summary and outlook.

2 An Experimental Vector Space Classifier

In this section, we introduce our experimental vector space classifier (VSC) system. The goal was to build an extremely high-performance classifier, in terms of accuracy, speed, and extensibility for the *classification of files by type*. UNIX was considered as a sample file system. The classifier examines a UNIX file (a document) and assigns it to one of a set of predefined UNIX file types (classes). To date, the classifier is able to distinguish the 47 different file types illustrated in Figure 1. These 47 types were those that were readily available in our environment. Note that the classifier presented here can be generalized to most non-UNIX file systems. In the sequel, we briefly review the underlying *vector space model* [SWY75]. We focus on issues that are unique and novel in our particular implementation.

VSCs are created for a given classification task in two steps:

1. *Schema definition.* The schema of the classifier is defined by describing the names and features of all the classes one would like to identify. The features, say $f_1 \ldots f_m$, span an m-dimensional feature space. In this feature space, each document d can be represented by a vector of the form $v_d = (a_1, \ldots, a_m)$ where coefficient a_i gives the value of feature f_i in document d.
2. *Classifier training.* The classifier is trained with training data – a collection of typical documents for each class. The frequency of each feature f_i in all training documents is determined. For each class c_i, a centroid $v_{c_i} = (\overline{a}_1, \ldots, \overline{a}_m)$ is computed, whose coefficients $\overline{a}_i$ are the mean values of the extracted features of all training documents for that class.

Given a trained classifier with centroids for each class, classification of a document d means finding the "most similar" centroid v_c and assigning d to that class c. A commonly used *similarity measure* is the *cosine metric* [vR79]. It defines the distance between document d and class centroid c by the angle α between the document vector and the centroid vector, that is,

$$\mathrm{sim}(d,c) \;=\; \cos\alpha \;=\; \frac{v_d \cdot v_c}{|v_d||v_c|}\,.$$

Building centroids from training data and using the similarity measure allows for very fast classification. To give a rough idea, an individual document can be classified by our system in about 40 milliseconds on an IBM RISC System/6000 Model 530H. In Section 2.2, we will compare the accuracy of the cosine metric with common alternatives.

2.1 Defining the Classifier's Schema

A feature is some identifiable part of a document that distinguishes between document classes. For example, a feature of LaTeX files is that file names usually have the extension "`.tex`" and that the text frequently contains patterns like "`\begin{...}`".

Features can either be boolean or counting. Boolean features simply determine whether or not a feature occurred in the document. Counting features determine how often a feature was detected. They are useful to partially filter out "noise" in the training data. Consider for example CSource files, having a lot of curly

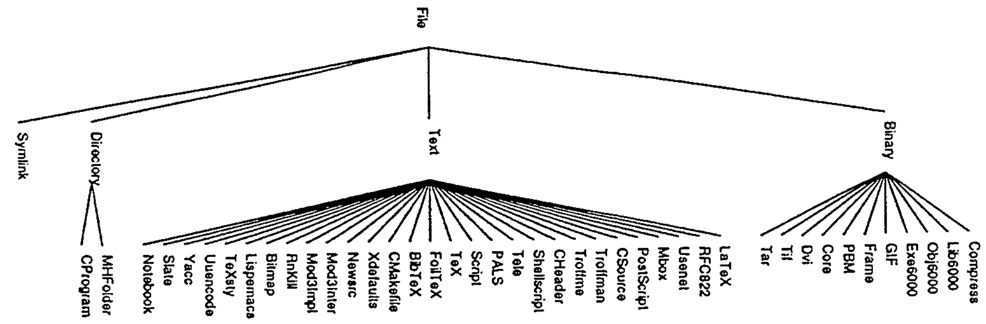


Figure 1: Class hierarchy of the experimental file classifier

braces. Many other file types have curly braces too, but only a few. Hence, counting this feature instead of just noting its presence would differentiate CSOURCE files from others.

In the remainder of this paper, we assume boolean features only. Using counting features is part of future work. In general, counting features in a VSC must be normalized before using, like proposed for example by the non-binary independence model [YMP89].

To define the schema of the file classifier, four different types of features are supported. They can be used to describe patterns that characterize file types:

- A `filenames` feature specifies a pattern that is matched against the name of a file;
- A `firstpats` feature specifies a pattern that is matched against the first line of a file;
- A `restpats` feature specifies a pattern that is matched against any line of a file;
- An `extract` feature specifies that there exists an extraction function (e.g. a C program) to determine if the feature is present.

The feature types `filenames`, `firstpats`, and `restpats` are processed by a pattern matcher. For performance reasons, this is a finite state machine specially built from the classifier schema. Patterns can either be string literals or regular expressions. The regular expressions supported are similar to the regular expressions of the UNIX "ed" command. The feature type `extract` is used to define file properties that cannot be described by regular expressions. For instance, `extract` features can be programmed to check whether a document is an executable file or a directory.

Any feature can be defined as `must`, which means that its occurrence is mandatory. If such a feature is not present in a given file, the file cannot be a member of that class. Notice that the converse is not true: the presence of a `must` feature does not force a type match.

Example 1: Figure 2 shows an excerpt of a sample classifier schema, defining classes for POSTSCRIPT pictures, LATEX documents, MHFOLDER directories, and COMPRESS files.

A `filenames` feature specifies that names of POSTSCRIPT files usually end with the extension "`.ps`", names of LATEX files with "`.tex`", and names of COMPRESS files with "`.z`" or "`.Z`". They are all defined as regular expressions, as indicated by the keyword "regexp".

A `firstpats` feature is defined for POSTSCRIPT files. It is a regular expression, saying that the first lines of these files always begin with "`%!`". This pattern is given with a `must` keyword, i.e., it must be present in POSTSCRIPT files.

`restpats` features specify that POSTSCRIPT files usually contain the two string literals "`%EndComments`" and "`%Creator`" and that LATEX files often contain the string literals "`\begin{`", "`\end{`", or "`{document}`".

Extraction functions exist for classes MHFOLDER and COMPRESS. Notice, that the implementation of extraction functions is not part of the classifier schema. However, by naming convention, they are implemented by C functions called "`ex_MHFolder`" and "`ex_Compress`" respectively. For example, "`ex_Compress`" searches for a file checksum and "`ex_MHFolder`" opens the directory and looks for mail files. ◇

Finding appropriate features for each class is crucial to the accuracy of a classifier [Jam85]. This has been verified by our experiments. For example, to define the 47 classes of the UNIX file classifier, a total of 206 features were carefully specified. We come therefore back to the issue of feature selection in Section 4.

2.2 Alternative Similarity Metrics

Diverse similarity metrics are proposed in the literature. For example, [vR79] describes Asymmetric, Cosine, Dice, Euclidian, Jaccard, and Overlap distance.

Table 1 summarizes our extensive classifier performance experiments. Experiments involved choosing random subsets from a collection of 26MB of sample

```
PostScript {
      filenames {
            "\.ps$" regexp
      }
      firstpats {
            "^%!" regexp must
      }
      restpats {
            "%EndComments"
            "%Creator:"
      }
}

MHFolder {
      extract
}
```

```
LaTeX {
      filenames {
            "\.tex$" regexp
      }
      restpats {
            "\begin{"
            "\end{"
            "{document}"
      }
}

Compress {
      extract
      filenames {
            "\.[zZ]$" regexp
      }
}
```

Figure 2: Sample classifier schema

data for training and then for performance testing. To find the closest centroid, the distance between document d and centroid c was alternatively measured with the above six common distance metrics (d_j, c_j means the j-th coefficient of the vector d or c respectively).[1]

Best and most reliable accuracy has been achieved using the cosine as similarity measure in our VSC. A promising alternative though is the asymmetric measure. It captures the inclusion relations between vectors, i.e., the more that properties of d are also present in c, the higher the similarity. Dice, Jaccard, and Overlap metrics give lower accuracy for our purposes. Surprisingly very low results have been achieved by Euclidian distance.

The bottom line of this evaluation is that the classifier's accuracy could not have been improved by choosing a different distance measure. In the following section we discuss a way of getting feedback about the classifier's confidence which can, in turn, be used to improve the accuracy of the classifier.

3 The Confidence Measure

Independent of which similarity measure is chosen, closeness to a centroid is not a very useful indicator of the classifier's confidence in its result. Hence, we introduce the following novel measure that gives important feedback on how sure the classifier is about a result.

Definition. The *confidence* of an assignment of document d to class c_i is defined as

$$\text{confidence}(d, c_i) \overset{\text{def}}{=} \frac{\text{sim}(d, c_i) - \text{sim}(d, c_j)}{\text{sim}(d, c_i)}$$

with c_i the closest centroid and c_j the second closest centroid.

The confidence is the ratio of the similarity of the closest and second closest centroid over the similarity of the file and the closest centroid.[2] The following example illustrates how the confidence measure works.

Example 2: Consider two centroids c_1 and c_2, having both the same distance from a given document d, i.e. $\text{sim}(d, c_i) = \text{sim}(d, c_j)$. Classification as one or the other class is therefore completely arbitrary.

However, if these centroids are very close to the document, the similarity alone suggests a very good classification result, which is not correct. The true situation is reflected by the confidence, which gives a very low value, namely 0. ◇

The confidence measure can be used to tell whether the classifier probably misclassified a document. The

[1] The accuracy of a classifier is measured for a particular class C as [Jon71]

$$\text{recall}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects of } C}$$

$$\text{precision}(C) = \frac{\text{objects of } C \text{ assigned to } C}{\text{total objects assigned to } C}\,.$$

To measure a classifier as a whole, we use the arithmetic mean of recall or precision over all classes. Notice that every object is classified into exactly one class (no unclassified or double classified objects). The E-value [vR79]

$$\text{E-value} = \frac{2 \text{ precision recall}}{\text{precision} + \text{recall}}$$

is a single measure of classifier accuracy that combines and equally weights both, recall and precision.

[2] The confidence measure can be generalized to take into account the n closest centroids. In this paper however, we use the closest and second closest centroids only.

Table 1: Alternative similarity metrics

distance metric	sim(d, c)	recall	precision	E-value
1. Cosine	$\frac{d \cdot c}{\lvert d\rvert\lvert c\rvert} = \cos\alpha$	0.97	0.97	0.97
2. Asymmetric	$\frac{\sum_j \min(d_j\ ,\ c_j)}{\sum_j d_j}$	0.94	0.95	0.95
3. Dice	$\frac{2\ (d \cdot c)}{\sum_j d_j + \sum_j c_j}$	0.94	0.93	0.94
4. Jaccard	$\frac{d \cdot c}{\sum_j d_j + \sum_j c_j - \sum_j (d_j\ c_j)}$	0.94	0.93	0.94
5. Overlap	$\frac{d \cdot c}{\min(\sum_j d_j\ ,\ \sum_j c_j)}$	0.93	0.90	0.91
6. Euclidian	$\sqrt{\sum_j (d_j - c_j)^2}$	0.69	0.87	0.77

higher the confidence value, the higher the classifier's certainty and therefore the higher the probability that the file is classified correctly.

Figure 3 shows the distribution of the confidence for a sample classifier. Each dot represents one of the ~2500 test files. The (logarithmic) x-axis shows the classifier's confidence in assigning a test file to a file type. The y-axis is separated into two areas, the lower one for correctly classified files and the upper one for incorrectly classified files. Both areas have one row for each of the 47 file types.

This distribution illustrates the tendency of correctly classified files to have a confidence around 0.7 and the incorrectly classified files around 0.07. One can make use of that to alert a human expert, that is, to apply the following algorithm: choose a *confidence threshold* Θ; classify document d, resulting in a class c with confidence γ; if $\gamma < \Theta$ then ask a human expert to approve the classification of document d as class c.

Figure 4 illustrates how much feedback can be derived from the confidence measure. Assumes a given confidence threshold Θ (vertical line), such that the user has to approve the classification if a file is classified with a smaller confidence.

The dotted curve shows the percentage of test files for which the assumption is true that they are classified correctly if classified with a confidence above threshold Θ and classified incorrectly otherwise. If, for example, the threshold Θ is set to 0.1, then about 94% are classified correctly if their confidence is above 0.1 and incorrectly otherwise (see dotted line hitting threshold).

The solid curve shows the percentage of test files that were classified with a confidence below Θ. With $\Theta = 0.1$, about 10% of the files are presented to the user for checking (see solid curve hitting the threshold). These were shown to a human expert. Note that about 5% of the files had a confidence of 0. These files were equidistant from 2 centroids indicating that the classifier had to make an arbitrary choice between them.

Finally, the dashed curve shows the percentage of test files that were classified correctly even though they have a confidence below threshold Θ. These are the files where the classifier "annoyed" the user for no good reason. With $\Theta = 0.1$, only 30% of the presented files were actually classified correctly (see dashed line hitting the threshold). Thus, using the confidence measure, a user had to touch 10% of all files, of which in fact 70% were classified incorrectly. The classifier's overall recall could therefore be improved by 7% without bothering the user too much.

In this classifier, $\Theta = 0.1$ provides a maximum accuracy (dotted line) while providing a reasonable number of files for the user's consideration while maintaining a modest "annoyance" level.

3.1 Classifier Training Strategies

The confidence measure's primary use is to detect misclassified documents. This not only improves the classifier's performance, but also proved to be useful for other purposes. In this section, we concentrate on using the confidence measure to speed up classifier training. Quick (re)training is an ability that is crucial for any classifier, especially for extensibility, as we will see later.

To train the classifier, a human expert has to provide a reasonable number of documents that are typical of each class. The first questions is: how much training data is required for it to perform well? Preliminary experiments showed that a surprisingly small set of training data produces a sufficiently accurate classifier. In Figure 5, the solid line shows the performance (E-value) of a classifier built with different sizes of training data.

For example, a classifier trained with only one document per class has average E-value of 0.89. The same

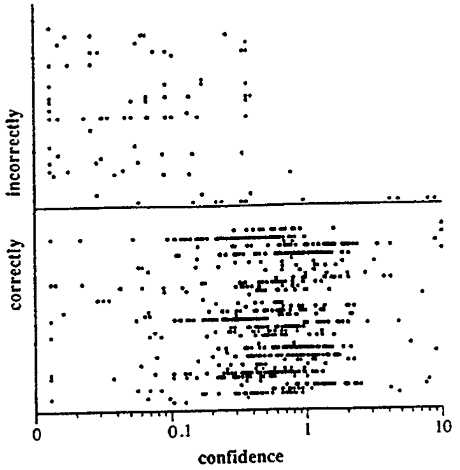


Figure 3: Distribution of the confidence measure

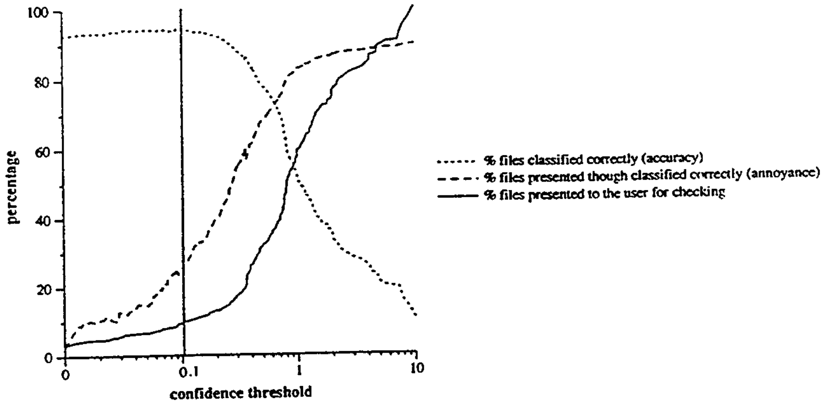


Figure 4: Feedback from the confidence measure

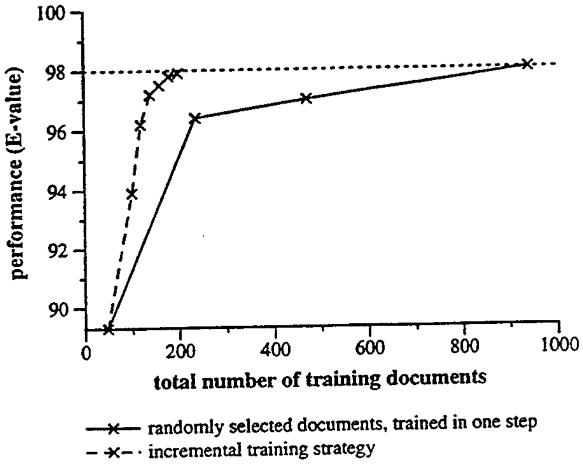


Figure 5: Number of training documents vs. classifier performance

classifier with 5 training documents has average E-value of about 0.96, with 10 documents about 0.97, and with 20 documents (~1000 total) nearly 0.98.

In the experiments discussed in the remainder of this paper, we use (unless stated otherwise) training data sets with an average of ~17 documents per class (total ~800 documents = ~26 MBytes). These data sets have randomly been selected as subsets of a large collection of training documents. On an IBM RISC System/6000 Model 530H, training the experimental file classifier takes about 50 seconds, with this amount of training data. For evaluating the classifier's performance randomly selected data is used that is always disjoint from the training data.

Though it shows that only little training data is required, the second question is: what are good training documents and how can they be found. One common way is to use an *incremental training strategy*, where the classifier is initially trained with few (one or two) documents of training data for each class. Then the classifier is run on unclassified test documents. A human expert manually classifies some of them and adds them to the training data. After about 20 documents have been added to the training data, the classifier is retrained with the extended training set.

The crucial parameter of this strategy is whether the correctly or the incorrectly classified documents should be added to the training data set. We actually used a third approach and added those documents to the training data for which the classifier was least confident about the classification, i.e., the confidence measure was below a given threshold. The final incremental training algorithm is illustrated in the following:

```
step 1:
   train an initial classifier with $N_0$
            documents per class;
step 2:
   while the classifier's performance is
            insufficient
   and a user is willing to classify
      documents do
      classify document using current
         classifier;
      if confidence was below a certain threshold
         then
         classify document by user and
         add it to training data set;
      if $N_1$ training documents have
         been added then
         retrain the classifier with new
         training set;
   end
```

Incremental training is very efficient when adding the least confident documents to the training data set. Consider again Figure 5: the dashed line shows the classifier's performance using the incremental training strategy, as opposed to training the classifier with randomly selected data, all at once (solid line). An initial classifiers was built with $N_0 = 2$ training documents per class (~100 documents in total), which resulted in an E-value of about 0.94. In five iterations, $N_1 = 20$ documents per iteration were incrementally added to the training data.

To achieve a classifier of E-value 0.96, one iteration was necessary. Notice, that at this point of time, only a total of 120 training documents were used, compared to 250 needed documents if training with random data in one step. After five iterations we already achieved 0.98 and used only 200 training documents, compared to 1,000 if trained with random data in one step (cf. Figure 5).

The incremental training algorithm is similar to the uncertainty strategy proposed by [LG94]. However, the number of files needed by their strategy is significantly larger than ours (up to 100,000 documents), because they are doing semantic full textual analysis of all the words in the documents. In contrast, we look for a few syntactic patterns and can get enough randomness in 10 files.

4 Feature Selection

Finding good features is crucial for a classifier's performance. However, it is a difficult task that can not be automated.

On one hand, features must identify one specific class and should apply as little as possible to other classes. This is easy for classes that can be identified by examining files for matching string literals, like e.g., FrameMaker documents, or GIF pictures. But it is difficult for classes that are very similar, like different kinds of electronic mail formats, e.g. RFC822 mail, Usenet messages, MBox folders. It may also be a problem for textual files containing mainly natural language and having only few commonalities.

On the other hand, there must be enough features to identify all kinds of files of a particular class. This causes a problem, if classes can only be described by very general patterns or can take alternative forms, like for instance word processors having different file saving formats. In these cases, it is advisable to either define completely different classes or to combine features together.

In this section, we present techniques to analyze and improve the schema of a classifier. These techniques help a human expert choose good features. To reveal the results in advance, we managed to improve a classifier's performance from an average E-value of 0.86 to 0.94, just by optimizing the schema.

4.1 Distinguishing Power

The most important property of features is how precise they identify one particular class. Thus, good features can be separated from bad features in how distinguishing they are, i.e., the number of classes they match.

We use the variation of feature coefficients over all centroids to measure how distinguishing features are. Consider vector $f_i = (a_{i1}, \ldots, a_{in})$, where a_{ij} is the coefficient of feature f_i in centroid c_j $(1 \leq i \leq m, 1 \leq j \leq n)$. This vector represents the feature's distribution over classes. Assuming normal distribution, we define:

Definition. The *distinguishing power* of feature f_i is defined as

$$\text{dist}-\text{power}(f_i) \overset{\text{def}}{=} \frac{s^2}{\overline{a}}$$

where $s^2 = \frac{1}{n-1}\sum_{j=1}^{n}(a_{ij} - \overline{a})^2$ is the variance and $\overline{a} = \frac{1}{n}\sum_{j=1}^{n} a_{ij}$ is the mean.

This definition of distinguishing power values both, low variance and low mean. It ranges from 0 to 1. For an optimal feature that has all coefficients $a_{ij} = 0$ except for one that is 1, the variance s^2 is equal to its mean $\overline{a}$. Hence, the distinguishing power of a perfect feature is 1. For a worst-case feature that has a uniform distribution over all classes (and $\overline{a} \neq 0$), the variance, and therefore the distinguishing power, is 0. The higher dist−power(f_i) is, the more distinguishing is feature f_i.

Example 3: Figure 6 illustrates distinguishing power for two sample features. Feature "Shellscript__set_" is defined for class ShellScript and searches for string literal " set ". This feature matches many different classes to a low degree, which is reflected in a very low distinguishing power (0.1978).

Feature "RFC822_^From:" is defined for class RFC822 (an e-mail format) and searches for lines beginning with "From:". This feature has a much better distinguishing power (0.7742). It selects fewer classes, most of them to a high degree. Notice that this feature now identifies a group of four classes that are similar (e-mail like).

An example of a perfect feature (distinguishing power 1.0) is feature "CHeader_.h$" (not shown in Figure 6), a regular expression looking for file names ending with ".h". Its coefficients are 1 for class CHeader and 0 for all others. ◇

In general, feature analysis can be performed in two different ways. These approaches are complementary:

- the analyzer scans human generated features and identifies those with poor distinguishing power;
- the analyzer scans all training documents and proposes features with high distinguishing power.

A human expert is necessary in both cases. Ultimately, the expert must decide whether to include a proposed feature into a schema, change an existing feature's definition in order to make it more specific, delete a feature, or keep it as it is. It is difficult to automate this task. Some features must be included although they are not very distinguishing, for instance, those that are the only feature of a top-level class in the hierarchy (Text, Binary, Directory, Symlink). On the other hand, regular expression patterns, for example, may contain an error that cannot be detected and corrected automatically.

To illustrate feature analysis, the experimental file VSC was built using a non-optimized schema with about 200 features, created by a user with moderate experience in using the classifier. This classifier had an average E-value of 0.86.

Subsequent feature analysis showed that only about 15% of these features identified exactly one type (dist-power = 1), 10% did not match any type at all, and more than 50% had dist-power < 0.5. Based on this feature analysis, the schema was optimized. Patterns were changed to make features more specific and syntax errors that caused features to fail to identify any class were corrected. A new classifier was built with this improved schema. The average E-value increased to 0.94, just from using the optimized schema.

4.2 Combining Features

Some file types have the property that documents of these classes match a highly varying number of features (e.g. Script, TroffMe, Yacc, CSource). Some documents match 20 to 30 features, whereas others only 1 or 2. Even if the 1 or 2 features are a subset of the 20 to 30 features, the classifier performs poorly for these classes, because it can only be trained to properly recognize one of the two styles of documents.

One approach would be to define two different classes to cover the two styles. However, it proved to be extremely difficult to define the schemas for the two separated classes and to separate the training data.

A better solution is combining several features $f_1, \ldots, f_m$ into one feature. The new feature is built as a regular expression $f = f_1 | \ldots | f_n$, connecting the

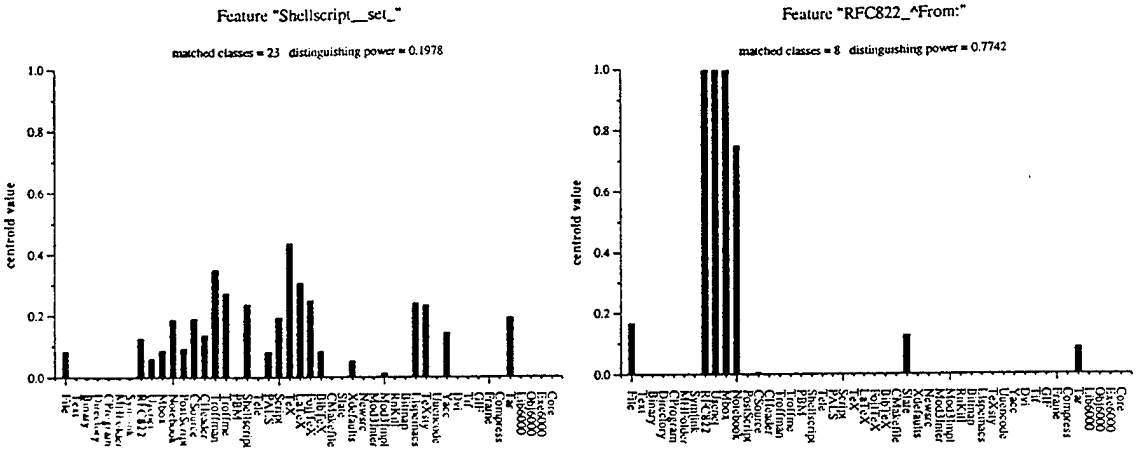


Figure 6: Distinguishing power of features

original features via "or" patterns. There are two ways to combine the features of a given class together:

- *Combining "disjoint" features.* The first way is to combine "disjoint" features that never (less than a given amount of the time) appear together. Consider as an example file types with two different initialization commands where only one of which appears at the beginning of the file.
- *Combining "duplicate" features.* The second way is to combine "duplicate" features, that is, features that always (more than a given amount of the time) appear together, but do not appear often (in more than a given amount of the files). For example, patterns "argc" and "argv" in CSource. The second limitation allows the classifier to keep the really good features like "`^Received`" and "`^From`" which appear in all RFC822 files, but it will combine "argc" and "argv" which only sometimes occur in CSource files.

The algorithm for combining features looks as follows (choosing 80% as the threshold to combine features and 60% for the number of files duplicate features should not appear in has given the best results):

```
foreach class $c_i$ ($i = 1 \ldots n$) of the schema do
step 1:
    $m$ = number of features of class $c_i$;
    $F$ = features $\{f_1, \ldots, f_m\}$ of class $c_i$;
    $P(F)$ = the powerset of $F$,
        without the empty set;
step 2:
    train the classifier:
    scan all training documents of class $c_i$
        for feature occurrences;
    foreach $s \in P(F)$ do
        $occ(s)$ = percentage of training
            documents of class $c_i$
            where features $s$ occur together;
    end
step 3:
    find features to be combined:
    while $m > 1$ do
        foreach $s \in P(F)$ with $m$ features do
            if $(\mathrm{avg}_{f \in s} occ(s)/occ(f) < 0.20)$
            or $((\mathrm{avg}_{f \in s} occ(s)/occ(f) > 0.80)$
            and $occ(s) < 0.60)$
            then
                combine features in $s$;
                remove all sets from $P(F)$
                containing any of the
                features in $s$;
        end
        $m - -$;
    end
end
```

The algorithm works class by class and combines only features that are defined within the same class, that is, features from different classes are never combined together.[3]

In step 1, the algorithm computes $P(F)$ as the set of all possible subsets of features $\{f_1, \ldots, f_m\}$ for the current class c_i. In step 2, the classifier is trained by classifying a large number of documents (~40 - 50 per class). While scanning training documents, the algorithm remembers for each of the feature combinations $s \in P(F)$ the percentage of documents in which this combination occurred. In step 3, the algorithm searches features to be combined. It tries to combine as many features as possible and starts therefore with the largest feature combination having all m features. If the combination fulfills one of the "disjoint" or "du-

[3] In the current experimental classifier, feature combination runs on restpats features only.

plicate" occurrence conditions, all features of the set s are removed from the schema and replaced by one new feature that combines them as described above. All sets are removed from $P(F)$ containing any of the features from s, because maximum combination was already achieved for these features. If all feature combinations of this size m have been processed, m is decremented and the algorithm tries to combine the remaining feature combinations of the smaller size.

Running feature combination on the `restpats` of the already optimized schema of the previous section, combined 37 disjoint features into 14 new features and 58 duplicate features into 20 new ones. Just by automatic feature combination, the classifier's performance has been improved from an overall E-value of 0.94 to now 0.96. In detail, the recall of file type TROFFME has been increased by 9.1% combining 7 into 2 features, YACC by 7.1% combining 2 into 1 feature and SCRIPT by 6.2% combining 38 into 13 features.

5 Comparison of other Classifier Technologies

There are diverse technologies for building classifiers. *Decision tables* are very simple classifiers that determine according to table entries what class to assign to an object. The UNIX "file" command is an example of a decision table based file classifier. It scans the `/etc/magic` file, the decision table, for matching values and returns a corresponding string that describes the file type. *Decision tree classifiers* construct a tree from training data, where leaves indicate classes and nodes indicate some tests to be carried out. CART [BFOS84] or C4.5 [Qui93] are well known examples of generic decision tree classifiers. *Rule based classifiers* create rules from training data. R-MINI [Hon94] is an example of a rule based classifier that generates disjunctive normal form (DNF) rules. Both, decision tree and rule classifiers, usually apply pruning heuristics to keep these trees/rules in some minimal form. *Discriminant analysis* (linear or quadratic) is a well known basic statistical classification approach [Jam85].

To rate our experimental vector space classifier, we built alternative file classifiers using quadratic discriminant analysis, decision tables, the decision tree system C4.5, and the rule generation approach R-MINI. Table 2 summarizes the conducted experiments. The intent of this table is to give a rough overview of how the different techniques compare on the file classification problem and not to present detailed performance numbers ("+" means an advantage and "-" means a disadvantage of a particular classifier technology).

Speed. Training and classification using quadratic discriminant analysis is very slow because extensive computations must be performed. All other classifier technologies provide fast training and classification. The vector space classifier simply needs to compute angles between the document vector and all centroids. For example, on an IBM RISC System/6000 model 530H, an individual document is classified by the experimental classifier in about 40 milliseconds, on average. The other classifier technologies (except of discriminant analysis) proved a similar speed.

Accuracy. Quadratic discriminant analysis and decision tables did not achieve our accuracy requirements. They had error rates up to 30%. All other classifier technologies proved much lower error rates. The C4.5 file classifier showed error rates from 2.6 to 5.0% misclassified files. The R-MINI file classifier showed error rates from 2.6 to 4.9%. The vector space classifier had 2.1 to 3.1% error rates. Hence, all three technologies have approximately the same range of errors.

Extensibility. A classifier is usually trained with a basic set of general classes. However, this basic class hierarchy must be extensible. Users want to define and add specific classes according to their personal purposes. *Extensibility* of a classifier is therefore crucial for many applications. In order to add classes to a classifier, a user must provide class descriptions (schema) and training data for the new classes. Training documents of the existing classes must be available too. This "old" training data is necessary because new classes must be trained with data for existing classes as well.

Vector space classifiers are highly suited for extensibility purposes. Consider as an example a classifier with existing classes $c_1, \ldots, c_g$ and existing features $f_1, \ldots, f_j$. Assume this classifier is extended with new classes $c_h, \ldots, c_n$ $(h = g + 1)$ and new features $f_k, \ldots, f_m$ $(k = j + 1)$. After extension, the *feature-centroid matrix* A of the classifier, where each coefficient a_{xy} shows the value of feature y in the centroid of class x, looks as follows:

$$\begin{array}{ccc} & \begin{array}{cccccc} f_1 & \cdots & f_j & f_k & \cdots & f_m \end{array} & \\ A = & \left[\begin{array}{ccc:ccc} a_{11} & \cdots & a_{1j} & a_{1k} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{g1} & \cdots & a_{gj} & a_{gk} & \cdots & a_{gm} \\ \hdashline a_{h1} & \cdots & a_{hj} & a_{hk} & \cdots & a_{hm} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nj} & a_{nk} & \cdots & a_{nm} \end{array}\right] & \begin{array}{c} v_{c_1} \\ \vdots \\ v_{c_g} \\ v_{c_h} \\ \vdots \\ v_{c_n} \end{array} \end{array}$$

The upper-left (dashed) sub-matrix of A shows the existing feature centroid matrix. To add new classes, these existing centroid coefficients need not be recomputed. The feature-centroid matrix can incrementally be extended with coefficients for newly added classes.

Table 2: Different Classifier Technologies

	Quad. Discr. Analysis	Decision Tables	Decision Trees (C4.5)	DNF Rules (R-MINI)	Vector Space Model
Speed	-	+	+	+	+
Accuracy	-	-	+	+	+
Extensibility	-	-	-	-	+

The lack of extensibility of discriminant analysis, decision table/tree and rule classifiers is the most dramatic difference. In contrast to vector space classifiers, extending this kind of classifiers with new user-specific classes demands rebuilding the whole system (tables, trees, or rules) from scratch, that is, it requires complete reconstruction of the classifier. Incremental, additive extension is not possible.

6 Conclusion and Outlook

High accuracy, fast classification, and incremental extensibility are the primary criteria for any classifier. The experimental VSC for assigning types to files presented in this paper fulfills all three requirements.

We evaluated different similarity metrics and showed that the cosine measure gives best results. A novel confidence measure was introduced that detects probably misclassified documents. Based on this confidence measure, an incremental training strategy was presented that significantly decreases the number of documents required for training, and therefore, increases speed and flexibility. The notion of distinguishing power of features was formalized and an algorithm for automatic combining disjoint and duplicate features was presented. Both techniques increase the classifier's accuracy again. Finally, we compared the VSC with other classifier technologies. It revealed that using the vector space model gives highly accurate and fast classifiers while it provides at the same time extensibility with user-specific classes.

The file classifier can be seen as a component of object. text, and image database management systems. There is recently an increasing interest in merging the functionality of database and file systems. Several proposals have been made, showing how files can benefit from object-oriented technology.

Christophides et al. [CACS94] describe a mapping from SGML documents into an object-oriented database and show how SGML documents can benefit from database support. Their work is restricted to this particular document type. It would be interesting to see how easily it can be extended to a rich diversity of types by using our classifier.

Consens and Milo [CM94] transform files into a database in order to be able to optimize queries on those files. Their work focuses on indexing and optimizing. They assume that files are already typed before reading, for example, by the use of a classifier.

Hardy and Schwartz [HS93] are using a UNIX file classifier in Essence, a resource discovery system based on semantic file indexing. Their classifier determines file types by exploiting naming conventions, data, and common structures in files. However, the Essence classifier is decision table based (similar to the UNIX "file" command) and is therefore much less flexible and tolerant.

The file classifier can also provide useful services in a next-generation operating system environment. Consider for instance a file system backup procedure that uses the classifier to select file-type-specific backup policies or compression/encryption methods.

Experiments have been conducted using the classifier for language and subject classification. Whereas language classification showed encouraging results, this technology has its limitations for subject classification. The reason is that the classifier works mainly by syntactical exploration of the schema, but subject classification must take into account the semantics of a document.

We are currently working on making the classifier extensible even without the requirement of training data for existing classes. We are also investigating the classification of structurally nested documents. A file classifier is being developed that is, for example, able to recognize Postscript pictures in electronic mail or C language source code in natural text documents. Use of this classifier to recognize, and take advantage, of a class hierarchy is an item for future work.

References

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In SIGMOD94 [SIG94b].

[CM94] M.P. Consens and T. Milo. Optimizing queries on files. In SIGMOD94 [SIG94b].

[GRW84] A. Griffiths, L.A. Robinson, and P. Willett. Hierarchic agglomerative clustering methods for automatic document classification. *Journal of Documentation*, 40(3), September 1984.

[Hoc94] R. Hoch. Using IR techniques for text classification. In SIGIR94 [SIG94a].

[Hon94] S.J. Hong. R-MINI: A heuristic algorithm for generating minimal rules from examples. In *Proc. of PRICAI-94*, August 1994.

[HS93] D.R. Hardy and M.F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proc. USENIX Winter Conf.*, San Diego, CA, January 1993.

[Jam85] M. James *Classification Algorithms.* John Wiley & Sons, New York, 1985.

[Jon71] K. S. Jones *Automatic Keyword Classification for Information Retrieval.* Archon Books, London, 1971.

[LG94] D.D. Lewis and W.A. Gale. A sequential algorithm for training text classifiers. In SIGIR94 [SIG94a].

[ODL93] K. Obraczka, P.B. Danzig, and S.-H. Li. Internet resource discovery services. *IEEE Computer*, 26(9), September 1993.

[Qui93] J.R. Quinlan *C4.5: Programs for Machine Learning* Morgan Kaufman, San Mateo, CA, 1993

[Sal89] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, 1989.

[Sch93] P. Schäuble SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proc. 16th Int'l ACM SIGIR Conf on Research and Development in Information Retrieval*, Pittsburg, PA, June 1993. ACM Press.

[SIG94a] *Proc. 17th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, Dublin, Ireland, July 1994. Springer.

[SIG94b] *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Minneapolis, Minnesota, May 1994. ACM Press.

[SLS$^+$93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proc. 19th Int'l Conf. on Very Large Data Bases (VLDB)*, Dublin, Irland, August 1993.

[SWY75] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11), November 1975.

[vR79] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 1979.

[YMP89] C.T. Yu, W. Meng, and S. Park. A framework for effective retrieval. *ACM Trans. on Database Systems*, 14(2), June 1989.

Don't Scrap It, Wrap It!
A Wrapper Architecture for Legacy Data Sources

Mary Tork Roth
IBM Almaden Research Center
torkroth@almaden.ibm.com

Peter Schwarz
IBM Almaden Research Center
schwarz@almaden.ibm.com

Abstract

Garlic is a middleware system that provides an integrated view of a variety of legacy data sources, without changing how or where data is stored. In this paper, we describe our architecture for wrappers, key components of Garlic that encapsulate data sources and mediate between them and the middleware. Garlic wrappers model legacy data as objects, participate in query planning, and provide standard interfaces for method invocation and query execution. To date, we have built wrappers for 10 data sources. Our experience shows that Garlic wrappers can be written quickly and that our architecture is flexible enough to accommodate data sources with a variety of data models and a broad range of traditional and non-traditional query processing capabilities.

1 Introduction

Most large organizations have collected a considerable amount of data, and have invested heavily in systems and applications to manage and access that data. It is increasingly clear that powerful applications can be created by combining information stored in these historically separate data sources. For example, a medical system that integrates patient histories, EKG readings, lab results and MRI scans would greatly reduce the amount of time required for a doctor to retrieve and compare these pieces of information before making a diagnosis.

Garlic is a middleware system that provides an integrated view of heterogeneous legacy data without changing how or where the data is stored. Middleware systems leverage the storage and data management facilities provided by legacy systems, providing a unified schema and common interface for new applications without disturbing existing applications. Freed from the responsibilities of storage and data management, these systems focus on providing powerful high-level query services for heterogeneous data.

Middleware systems typically rely on *wrappers* [4] [18] [9] that encapsulate the underlying data and mediate between the data source and the middleware. The wrapper architecture and interfaces are crucial, because wrappers are the focal point for managing the diversity of data sources. Below a wrapper, each data source, or *repository*, has its own data model, schema, programming interface, and query capability. The data model may be relational, object-oriented, or specialized for a particular domain. The schema may be fixed, or vary over time. Some repositories support a query language, while others are accessed using a class library or other programmatic interface. Most critically, repositories vary widely in their support for queries. At one end of the spectrum are repositories that only support simple scans over their contents (e.g., files of records). Somewhat more sophisticated repositories may allow a record ordering to be specified, or be able to apply certain predicates to limit the amount of data retrieved. At the other end of the spectrum are repositories like relational databases that support complex operations like joins or aggregation. Repositories can also be quite idiosyncratic, allowing, for example, only certain forms of predicates on certain attributes, or joins between certain collections. The wrapper architecture of Garlic [4] addresses the challenge of diversity by standardizing how information in data sources is described and accessed, while taking an approach to query planning in which the wrapper and the middleware dynamically determine the wrapper's role in answering a query.

This paper describes the Garlic wrapper architecture, and summarizes our experience building wrappers for ten data sources with widely varying data models and degrees of support for querying. The next section gives a brief overview of Garlic, and is followed by a section that summarizes the goals of the wrapper architecture. Section 4 describes how a wrapper is built, and Section 5 discusses the current status of our system. Section 6 briefly summarizes related work, and Section 7 concludes the paper and presents some opportunities for future research.

2 An Overview of Garlic

Garlic applications see heterogeneous legacy data stored in a variety of data sources as instances of objects in a unified

This work was partially supported by DARPA Contract F33615-93-1-1339.

Proceedings of the 23rd VLDB Conference Athens, Greece, 1997.

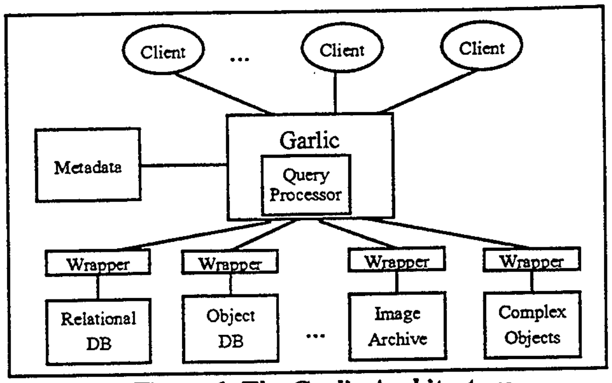


Figure 1. The Garlic Architecture.

schema. Rather than invent yet another object-oriented data model, Garlic's data model and programming interface are based closely on the Object Database Management Group (ODMG) standard [5]. Methods are of particular importance to Garlic, since they provide a convenient and natural way to model the specialized search and data manipulation facilities of non-traditional data sources. By extending SQL to allow invocations of such methods in queries, Garlic provides a single straightforward language extension that can support many different kinds of specialized search.

The overall architecture of Garlic is depicted in Figure 1. Associated with each repository is a wrapper. In addition to the repositories containing legacy data, Garlic provides its own repository for *Garlic complex objects,* which users can create to bind together existing objects from legacy repositories. Garlic also maintains global metadata that describes the unified schema. Garlic objects can be accessed both via a C++ programming interface and through Garlic's query language, an extension of SQL that adds support for path expressions, nested collections and methods. The heart of the Garlic middleware is the query processing component. The query processor develops plans to efficiently decompose queries that span multiple repositories into pieces that individual repositories can handle. The query execution engine controls the execution of such a query plan, by assembling the results from the repositories and performing any additional processing required to produce the answer to the query.

3 Goals for the Wrapper Architecture

Our experience in building wrappers for Garlic confirms that the architecture we describe in this paper achieves several goals that make it well-suited to integrate a diverse set of data sources. We summarize these goals here before describing the wrapper architecture in detail.

1. *The start-up cost to write a wrapper should be small.* We expect a typical Garlic application to combine data from several traditional sources (e.g., relational database systems from various vendors) with data from a variety of non-traditional systems such as image servers, searchable web sites, etc., and one-of-a-kind sources such as a home-grown molecular similarity search engine. Although Garlic is intended to ship with wrappers for popular data sources, we must rely on third party vendors and customer data administrators to provide wrappers for more specialized data sources. To make wrapper authoring as simple as possible, we require only a small set of key services from a wrapper, and ensure that a wrapper can be written with very little knowledge of Garlic's internal structure. In our experience, a wrapper that provides a base level of service can be written in a matter of hours. Even such a basic wrapper permits a significant amount of the repository's data and functionality to be exposed through the Garlic interface.
2. *Wrappers should be able to evolve.* Our standard methodology in building wrappers has been to start with a version that models the repository's content as objects and allows Garlic to retrieve their attributes. We then incrementally improve the wrapper to exploit more of the repository's native query processing capabilities.
3. *The architecture should be flexible and allow for graceful growth.* We require only that a data source have some form of programmatic interface, and we make no assumptions about its data model or query processing capabilities. Wrappers for new data sources can be integrated into existing Garlic databases without disturbing legacy applications, other wrappers, or existing Garlic applications.
4. *The architecture should readily lend itself to query optimization.* The author of a Garlic wrapper need not code to a standard query interface that may be too high-level or too low-level for the underlying data source. Instead, a wrapper is a full participant in query planning, and may use whatever knowledge it has about a repository's query and specialized search facilities to dynamically determine how much of a query the repository is capable of handling. This design allows us to build wrappers for simple data sources quickly, and still exploit the unique capabilities of unusual data sources such as image servers, text search engines, engines for molecular similarity search, etc.

4 Building a Garlic Wrapper

As shown in Figure 2, a wrapper provides four major services in the Garlic system. First, a wrapper models the contents of its repository as Garlic objects, and allows Garlic to retrieve references to these objects. Secondly, a wrapper allows Garlic to invoke methods on objects and retrieve their attributes. This mechanism is important, because it provides a means by which Garlic can get data out of a repository, even if the repository has almost no support for querying. Third, a wrapper participates in query planning when a Garlic query ranges over objects in its repository. The Garlic metadata does not include information about the query processing capabilities of individual repositories, so the Garlic query processor has no *a priori* knowledge about

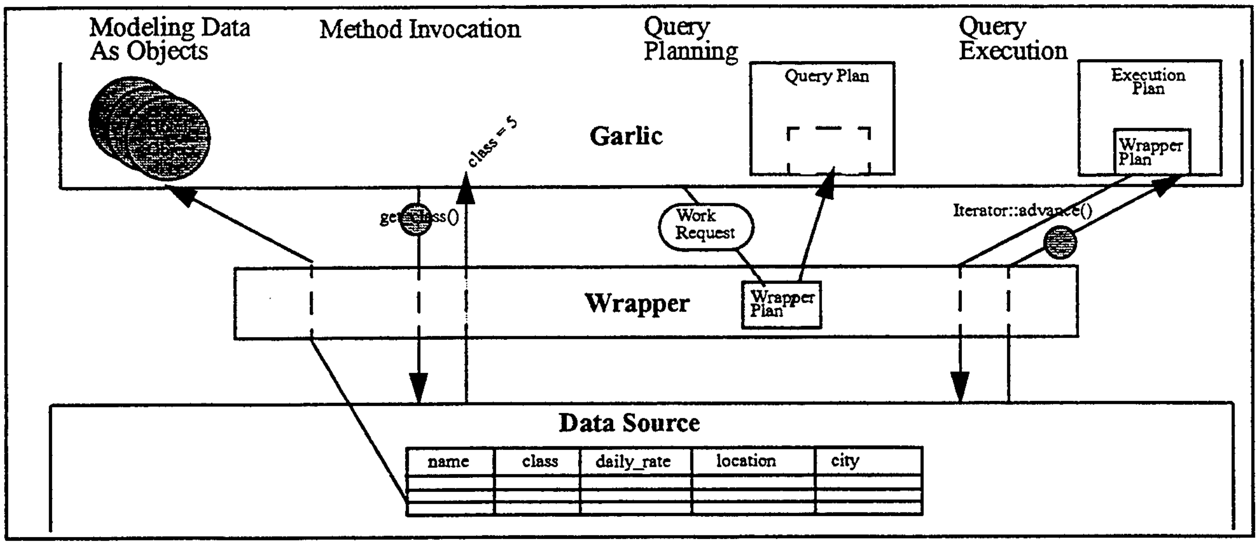


Figure 2. Services Provided by a Wrapper.

what predicates and projections can be handled by a given repository. Instead, the query processor identifies portions of a query relevant to a repository and allows the repository's wrapper to determine how much of the work it is willing to handle. The final service provided by a wrapper is query execution. During query execution, the wrapper completes the work it reported it could do in the query planning phase. A wrapper may take advantage of whatever specialized search facilities the repository provides in order to return the relevant data to Garlic.

In the sections that follow, we describe each of these services in greater detail, and provide an example of how to build wrappers for a simple travel agency application.

4.1 Modeling Data as Objects

The first service that a wrapper provides is to turn the data of the underlying repository into objects accessible by Garlic. Each Garlic object has an *interface* that abstractly describes the object's behavior, and an *implementation* that provides a concrete realization of the interface. The Garlic data model permits any number of implementations for a given interface. For example, two relational database repositories that contain information about disjoint sets of employees may each export distinct implementations of a common `Employee` interface.

During an initial registration step, wrappers provide a description of the content of their repositories using the Garlic Data Language, or GDL. GDL is a variant of the ODMG's Object Description Language (ODMG-ODL). The interfaces that describe the behavior of objects in a repository are known collectively as the *repository schema*. Repositories are registered as parts of a Garlic database and their individual repository schemas are merged into the *global schema* that is presented to Garlic users.

A wrapper also cooperates with Garlic in assigning identity to individual objects so that they can be referenced from Garlic and from Garlic applications. A Garlic object identifier (OID) has two parts. The first part, the *implementation identifier* (IID), is assigned by Garlic and identifies which implementation is responsible for the object, which in turn identifies the interface that the object supports and the repository in which it is stored. The second part of the OID, the *key*, is uninterpreted by Garlic. It is provided by the wrapper and identifies an object within a repository. Specific objects, usually collections, can be designated as *roots*. Root objects are identified by name, as well as by OID, and as such can serve as starting points for navigation or querying (e.g., root collection objects can be used in the `from` clause of a query).

As an example of how data is modeled as objects in Garlic, consider a simple application for a travel agency[1]. The agency stores information about the countries and cities for which it arranges tours as tables in a relational database. It also has access to a web site that provides booking information for hotels throughout the world, and to an image server in which it stores images of different travel destinations. These images can be retrieved and ordered according to features such as color, shape, texture, etc.

These sources are easily integrated as a Garlic database. The description of the `Country` and `City` interfaces that describe the relations in the relational database are shown in the left column of Figure 3. The attributes of each interface correspond to the columns of each relation, and the primary key value of a tuple serves as the key portion of the Garlic OID. Note that the `country` attribute on the `City` interface and the `scene` attributes on the `Country` and `City` interfaces are Garlic references to other Garlic objects. The relational wrapper registers `Cities` as a root collection of `City` objects, and `Countries` as a root collection of `Country` objects.

The web wrapper exports a single root collection of `Hotel` objects. The GDL for a `Hotel` object is shown at the

[1] For brevity, we have omitted many of the implementation details of this application. See [22] for a more precise description.

Relational Repository Schema

```
interface Country {
  attribute string name;
  attribute string airlines_served;
  attribute boolean visa_required;
  attribute Image scene;}

interface City {
  attribute string name;
  attribute long population;
  attribute boolean airport;
  attribute Country country;
  attribute Image scene;}
```

Web Repository Schema

```
interface Hotel {
  attribute readonly string name;
  attribute readonly short class;
  attribute readonly double daily_rate;
  attribute readonly string location;
  attribute readonly string city;}
```

Image Server Repository Schema

```
interface Image {
  attribute readonly string file_name;
  double matches(in string file_name);
  void display(in string device_name);}
```

Figure 3. Travel Agency Application Schema.

top of the right hand column in Figure 3. The web site provides unique identifiers on the HTML page for hotel listings it returns, and these identifiers serve as the key portion of `Hotel` OIDs.

The interface for the image data stored in the image server is provided at the bottom of the right hand column of Figure 3. The image server repository exports 2 methods on the `Image` interface: `matches()`, which takes as input the name of a file containing the description of an image feature and returns as output a score that indicates how well an image matches the feature, and `display()`, which models the server's ability to output an image on a specified device. Image file names provide the key for `Image` OIDs.

4.2 Method Invocation

The second service a wrapper provides is a means to invoke methods on the objects in its repository. Method invocations can be generated by Garlic's query execution engine (see Section 4.3), or by a Garlic application that has obtained a reference to an object (either as the result of a query or by looking up a root object by name).

In addition to explicitly-defined methods like `matches()`, two types of *accessor* methods are implicitly defined for retrieving and updating an object's attributes — a "get" method for each attribute in the interface, and a "set" method for attributes that are not read-only. For instance, a `get_class()` method would be implicitly defined for the read-only `class` attribute of the `Hotel` interface.

Garlic uses the IID portion of a target object's OID to route a method invocation to the object's implementation. The implementation must be able to invoke each explicitly defined method in the corresponding interface, as well as the accessor methods. An implementation consists of wrapper code that maps Garlic method invocations into appropriate operations provided by the repository. To accommodate the widest possible range of repositories, Garlic provides two variants of method invocation: *stub* and *generic* dispatch.

A wrapper that utilizes stub dispatch provides a stub routine for each method of an implementation. Stub dispatch is a natural choice for repositories whose native programming interface is a class library, such as the image server in our travel agency example. For the `display()` method, for example, the image server wrapper provides a routine that first extracts the file name of the target image from the key field of the OID, and unpacks the device name from the argument list supplied by Garlic. To display the image on the screen, the routine calls the appropriate display function from the image server's class library, giving the image file name and display name as arguments.

Generic dispatch is useful for repositories that support a generic method invocation mechanism, or for repositories that do not directly support objects and methods. A wrapper that supports generic dispatch exports a single method invocation entry point. An important advantage of generic dispatch is that it is schema-independent. A single copy of the generic dispatch code can be shared by repositories that have a common programming interface but different schemas. The relational wrapper is an example of a wrapper that uses generic method dispatch. This wrapper supports only accessor methods, and each method invocation translates directly to a query over the relation that corresponds to the target object's implementation. The wrapper maps the method name into a column name, maps the IID portion of the object's OID into a relation name, extracts the primary key value from the OID, and uses these values to construct a query to send to the database.

4.3 Query Planning

A wrapper's third obligation is to participate in query planning. The goal of query planning is to develop alternative plans for answering a query, and then to choose the most efficient one. The Garlic query optimizer [8] is a cost-based optimizer modeled on Lohman's grammar-like rule approach [12]. STARs (STrategy Alternative Rules) are used in the optimizer to describe possible execution plans for a query. The optimizer uses dynamic programming to build query plans bottom-up. First, single collection access plans are generated, followed by a phase in which 2-way join plans are considered, followed by 3-way joins, etc., until a complete plan for the query has been chosen. Garlic extends the STAR approach by introducing wrappers as full-fledged participants during plan enumeration. During each query planning phase, the Garlic optimizer identifies the largest possible query fragment that involves a particular repository, and sends it to the repository's wrapper. The wrapper returns zero or more plans that implement some or all of the work represented by the query fragment. The optimizer incorporates each wrapper plan into the set of plans it is considering to produce the results of the entire query, adding operators to perform in Garlic any portion of the query fragment that the wrapper did not agree to handle.

As noted previously, repositories vary greatly in their query processing capabilities. Furthermore, each repository has its own unique set of restrictions on the operations it will perform. These capabilities and restrictions may be difficult or impossible to express declaratively. For example, relational databases often have limits on the number of tables involved in a join, the maximum length of a query string, the maximum value of a constant in a query, etc. These limits vary for different products, and even for different versions of the same product. As another example, our web wrapper is able to handle SQL `LIKE` predicates, but is sensitive to the placement of wild card characters. A key advantage to our approach is that the optimizer does not need to track the minute details of the capabilities and restrictions of the underlying data sources. Instead, the wrapper encapsulates this knowledge and ensures that the plans it produces can actually be executed by the repository.

Our approach allows a wrapper to model as little or as much of the repository's capabilities as makes sense. If a repository has limited query processing power, then the amount of code necessary to support the query planning interface is small. On the other hand, if a repository does have specialized search facilities and access methods that Garlic can exploit, the interface is flexible enough for a wrapper to encapsulate as much of these capabilities as possible. Even if a repository can do no more than return the OIDs of objects in a collection, Garlic can evaluate an arbitrary query by retrieving data from the repository via method invocation and processing it within Garlic.

A wrapper's participation in query planning is controlled by a set of methods that the optimizer may invoke during plan enumeration. The `plan_access()` method is used to generate single-collection access plans, and the `plan_join()` method is used to generate multi-way join plans. Joins may arise from queries expressed in standard SQL, or joins may be generated by Garlic for queries that contain path expressions, a feature of Garlic's extended SQL. The `plan_bind()` method is used to generate a specific kind of plan that can serve as the inner stream of a bind join (to be described in Section 4.3.3). Each of these methods takes as input a *work request*, which is a lightweight parse-tree description of the query fragment to be processed. The return value is a set of plans, each of which includes a list of *properties* that describe how much of the work request the plan implements, and at what cost. The plans are represented by instances of a wrapper-specific specialization of a `Wrapper_Plan` class. In addition to the property list, they encapsulate any repository-specific information a wrapper needs to actually perform the work described by the plan.

4.3.1 Single Collection Access Plans

The `plan_access()` method is the interface by which the Garlic query optimizer asks a wrapper for plans that return data from a single collection. It is invoked for each collection to which a Garlic query refers. The work request for a single-collection access includes predicates to apply, attributes to project, and methods to invoke. Since the Garlic optimizer does not know *a priori* which (if any) of the predicates a wrapper will be able to apply, the projection list in a work request contains *all* relevant attributes and methods mentioned in the query, including those that only appear in predicates. This gives the wrapper an opportunity to supply values for attributes that the Garlic execution engine will need in order to apply predicates that the wrapper chooses not to handle. As a worst-case fallback, the projection list also always includes the OID, even if the user's original query made no mention of it. The execution engine uses the OID and the method invocation interface to retrieve the values of any attributes it needs that are not directly supplied by the wrapper.

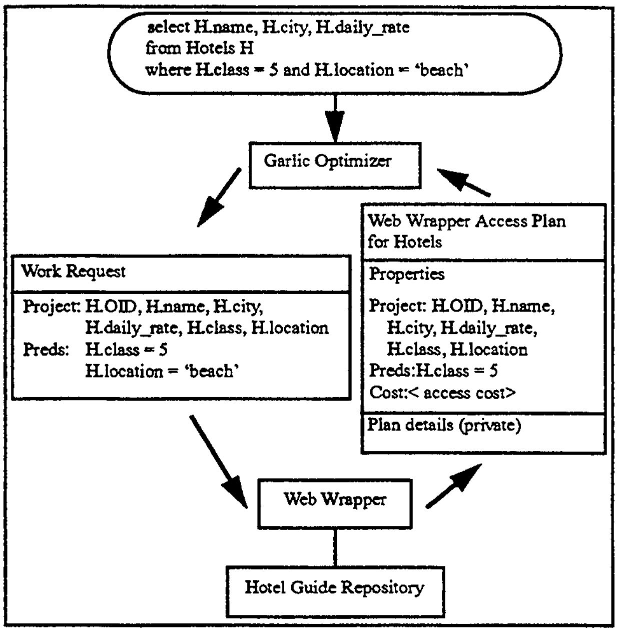

select H.name, H.city, H.daily_rate
from Hotels H
where H.class = 5 and H.location = 'beach'
Garlic Optimizer
Work Request
Project: H.OID, H.name, H.city, H.daily_rate, H.class, H.location
Preds: H.class = 5
H.location = 'beach'
Web Wrapper Access Plan for Hotels
Properties
Project: H.OID, H.name, H.city, H.daily_rate, H.class, H.location
Preds:H.class = 5
Cost:< access cost>
Plan details (private)
Web Wrapper
Hotel Guide Repository

Figure 4. Construction of a Wrapper Access Plan.

Figure 4 shows the first phase of query planning for a simple single-collection query against our travel agency database. Suppose a Garlic user submits a query to find 5-star hotels with beach front property. The Garlic query optimizer analyzes the user's query and identifies the fragment that involves the `Hotels` collection. Since the `Hotels` collection is managed by the web wrapper, it invokes the web wrapper's `plan_access()` method with a description of the work to be done. This description contains the list of predicates to apply and attributes to project.

During the execution of `plan_access()`, the web wrapper looks at the work request to determine how much of the query it can handle. In general, our web wrapper can project any attribute and will accept predicates of the form `<attr> <op> <const>`, where `<op>` is either = or the SQL keyword `LIKE`. However, the web wrapper cannot handle equality predicates on strings because the web site does not adhere to SQL semantics for string equality. The web site treats the predicate "`location = 'beach'`" as "`location LIKE '%beach%'`", which provides a superset of the results of the equality predicate. This differ-

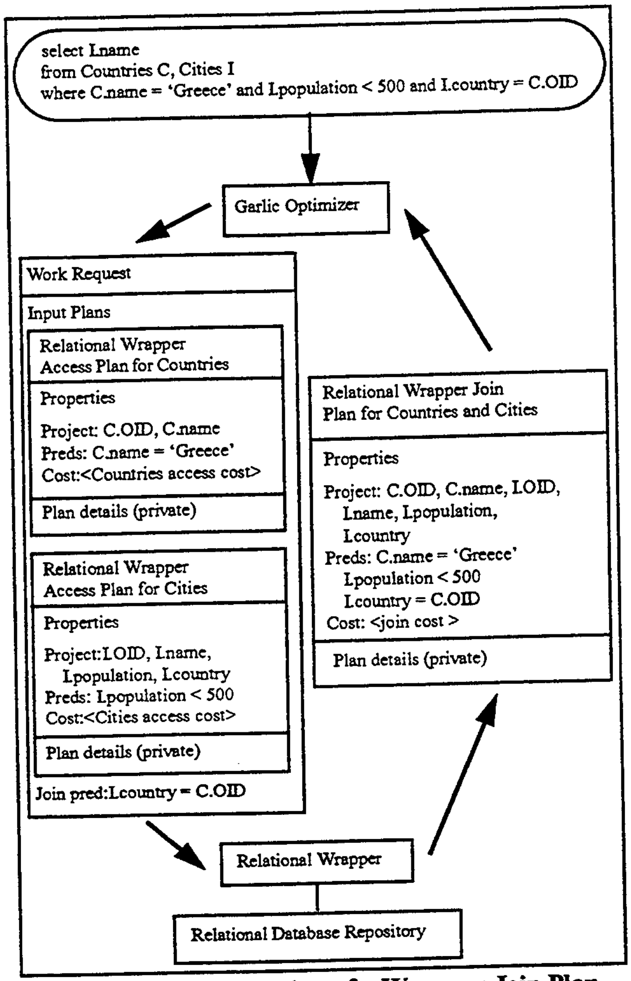

select I.name
from Countries C, Cities I
where C.name = 'Greece' and I.population < 500 and I.country = C.OID
Garlic Optimizer
Work Request
Input Plans
Relational Wrapper
Access Plan for Countries
Properties
Project: C.OID, C.name
Preds: C.name = 'Greece'
Cost:<Countries access cost>
Plan details (private)
Relational Wrapper
Access Plan for Cities
Properties
Project:I.OID, I.name, I.population, I.country
Preds: I.population < 500
Cost:<Cities access cost>
Plan details (private)
Join pred:I.country = C.OID
Relational Wrapper Join
Plan for Countries and Cities
Properties
Project: C.OID, C.name, I.OID, I.name, I.population, I.country
Preds: C.name = 'Greece'
I.population < 500
I.country = C.OID
Cost: <join cost >
Plan details (private)
Relational Wrapper
Relational Database Repository

Figure 5. Construction of a Wrapper Join Plan.

ence in semantics means that the web wrapper cannot report to the optimizer that it can apply a string equality predicate. Nevertheless, when string equality is requested, it is still beneficial for the wrapper to apply the less restrictive `LIKE` predicate in order to reduce the amount of data returned to Garlic. The wrapper therefore creates a plan that will handle the entire projection list, perform the predicate on `class`, and the predicate "`location LIKE '%beach%'`", while reporting through the plan's properties that the `location` predicate will not be applied.The wrapper assigns the plan an estimated cost and returns it to the optimizer. If this access plan is chosen to be part of the global plan for the user's query, the optimizer will need to add the necessary operator to apply the predicate on `location`, although it would be applied to a far smaller set of objects than if the wrapper had not (covertly) applied the `LIKE` predicate.

4.3.2 Join Plans

The Garlic query optimizer uses the access plans generated in the first phase of optimization as a starting point for join enumeration. If the optimizer recognizes that two collections reside in the same repository, it invokes the wrapper's `plan_join()` method (if one is implemented) to try to push the join down to that repository. The work request includes the join predicates as well as the single-collection access plans that the wrapper had previously generated for the collections being joined. In the `plan_join()` method, the wrapper can re-examine these plans, and consider the effect of adding join predicates.

Let's return to our travel agency. Figure 5 shows how the relational wrapper provides a plan for a join between `Countries` and `Cities`. In the first phase of optimization (omitted from the picture), the optimizer requested and received access plans for `Cities` and `Countries` from the relational wrapper. During join enumeration, the optimizer invokes the relational wrapper's `plan_join()` method and passes in the join predicate as well as the two access plans previously created. The wrapper agrees to perform all of the work from its original access plans and to accept the join predicate, and creates a new plan for the join.The new plan's properties are made up of the properties from the input plans and the new join predicate.

During the next phase of join enumeration, the optimizer will follow a similar procedure for 3-ways joins of collections that reside in the same repository, and so on.

4.3.3 Bind Plans

During the join enumeration phase, the Garlic optimizer also considers a particular kind of join called a *bind join,* similar to the fetch-matches join methods of [14] and [13]. In a bind join, values produced by the outer node of the join are passed by Garlic to the inner node, and the inner node uses these values to evaluate some subset of the join predicates. A wrapper is well suited to serve as the inner node of a bind join if the programming interface of its repository provides some mechanism for posing parameterized queries. As an example, ODBC and the call level interfaces of most relational database systems contain such support.

Suppose our travel agency user is really interested in finding 5-star hotels on beaches in small towns in Greece. This query involves the `Countries` and `Cities` collections managed by the relational wrapper, and the `Hotels` collection managed by the web wrapper. The web wrapper does not support the `plan_bind()` method, but the relational wrapper does. Figure 6 shows how a bind plan for this query is created. During the first phase of optimization, the optimizer would have requested and received an access plan from the web wrapper for the `Hotels` collection as described in Section 4.3.1. It would also have requested and received access plans for the `Countries` and `Cities` collections from the relational wrapper. While considering 2-way joins, the optimizer would have received a join plan for `Countries` and `Cities` from the relational wrapper, as described in the previous section.

Next, the optimizer develops a plan to join all three collections. The optimizer recognizes that a bind join is possible, with the web wrapper's access plan as the outer stream

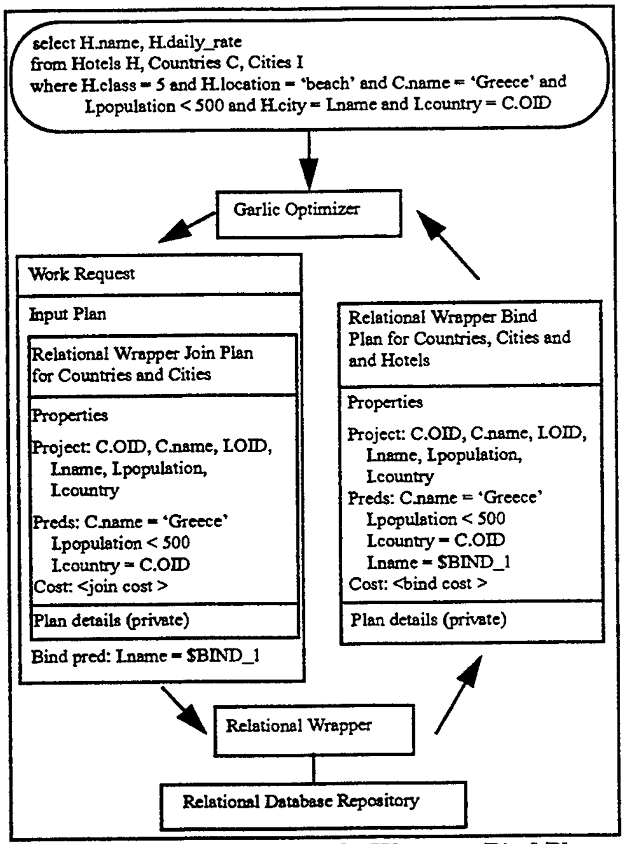

select H.name, H.daily_rate
from Hotels H, Countries C, Cities I
where H.class = 5 and H.location = 'beach' and C.name = 'Greece' and
I.population < 500 and H.city = I.name and I.country = C.OID
Garlic Optimizer
Work Request
Input Plan
Relational Wrapper Join Plan for Countries and Cities
Properties
Project: C.OID, C.name, I.OID, I.name, I.population, I.country
Preds: C.name = 'Greece' I.population < 500 I.country = C.OID
Cost: <join cost >
Plan details (private)
Bind pred: I.name = $BIND_1
Relational Wrapper Bind Plan for Countries, Cities and and Hotels
Properties
Project: C.OID, C.name, I.OID, I.name, I.population, I.country
Preds: C.name = 'Greece' I.population < 500 I.country = C.OID I.name = $BIND_1
Cost: <bind cost >
Plan details (private)
Relational Wrapper
Relational Database Repository

Figure 6. Construction of a Wrapper Bind Plan.

and the join plan provided by the relational wrapper as the inner stream. The optimizer invokes the relational wrapper's `plan_bind()` method, passing in a work request that consists of the join plan for `Countries` and `Cities` that the wrapper previously provided and the description of the bind join predicate between `Cities` and `Hotels`. The relational wrapper creates a new plan that handles the work of the original join plan plus the bind predicate. It uses the input plan's properties to fill in the new bind plan properties, and adds in the bind predicate.

4.4 Query Execution

A wrapper's final service is to participate in plan translation and query execution. A Garlic query plan is represented as a tree of operators, such as FILTER, PROJECT, JOIN, etc. Wrapper plans show up as the operators at the leaves of the plan tree. Figure 7 shows an example of a complete Garlic plan based on the bind join plan for the query discussed in Section 4.3.3. The outer node of the bind join is the web wrapper's access plan from Section 4.3.1, and the inner node is the relational wrapper's bind plan described in Section 4.3.3. The Garlic optimizer added a FILTER operator to handle the predicate on `location` and a PROJECT operator to project `name` and `daily_rate`.

The optimized plan must be translated into a form suitable for execution. As is common in demand-driven runtime systems [7], operators are mapped into iterators, and each wrapper provides a specialized `Iterator` subclass that controls execution of the work described by one of its plans. The wrapper must also supply an implementation of `Wrapper_Plan::translate()`, to translate a wrapper's plan into an instance of the wrapper's `Iterator` subclass. Translation involves converting the information stored in the plan into a form that can be sent to the repository. For example, our relational wrapper stores the elements of the `select`, `from` and `where` clauses of the query to be sent to the relational database in the private section of its plan. At plan translation time, the wrapper extracts these elements, constructs the query string, and stores it in an instance of its `Iterator` subclass. As another example, our web wrapper stores the list of attributes to project and the set of predicates to apply in the private data section of its plan. At plan translation time, the predicates are used to form a query URL that the web site will accept.

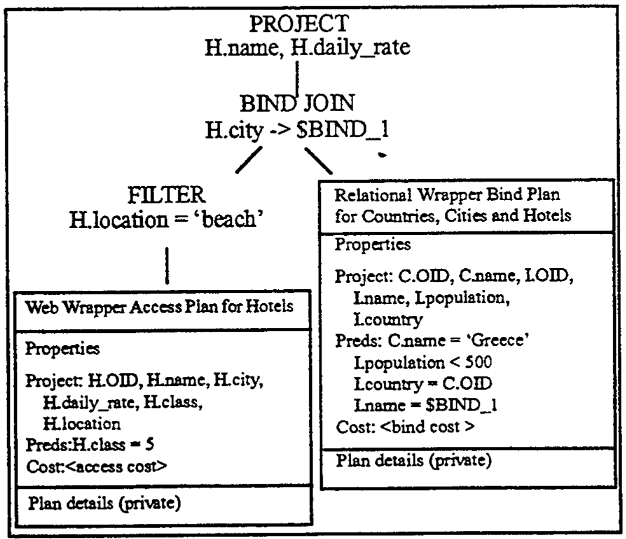

PROJECT
H.name, H.daily_rate
BIND JOIN
H.city -> $BIND_1
FILTER
H.location = 'beach'
Web Wrapper Access Plan for Hotels
Properties
Project: H.OID, H.name, H.city, H.daily_rate, H.class, H.location
Preds:H.class = 5
Cost:<access cost>
Plan details (private)
Relational Wrapper Bind Plan for Countries, Cities and Hotels
Properties
Project: C.OID, C.name, I.OID, I.name, I.population, I.country
Preds: C.name = 'Greece' I.population < 500 I.country = C.OID I.name = $BIND_1
Cost: <bind cost >
Plan details (private)

Figure 7. A Plan for a Garlic Query.

The Garlic execution engine is pipelined, and employs a fixed set of methods on iterators at runtime to control the execution of a query. Default implementations for most of the methods exist, but for each operator, two methods in particular define the unique behavior of its iterator: `advance()` and `reset()`. The `advance()` method completes the work necessary to produce the next output value, and the `reset()` method resets an iterator so that it may be executed again. An additional `bind()` method is unique to wrapper iterators, and provides the mechanism by which Garlic can transfer the next set of bindings to the inner node of a bind join.

Our relational wrapper uses standard ODBC calls to implement `reset()`, `advance()` and `bind()`. `reset()` prepares a query at the underlying database, and `bind()` binds the parameters sent by Garlic to the unbound values in the query string. The `advance()` method fetches the next set of tuples from the database.

The web wrapper's `Iterator` subclass is very simple. The `reset()` method loads the HTML page that corresponds to the query URL generated at plan translation time. In the `advance()` method, the wrapper parses the HTML page to extract the query results. Each HTML page pro-

TABLE 1. A Description of Existing Wrappers.

DB2, Oracle	Schema description: Columns of a relation map to attributes of an interface; relations become collections of objects; primary key value of a tuple is key for OID. Method invocation: accessor methods only, generic dispatch. Query operations: general expression projections, all basic predicates, joins, bind joins, joins based on OID.
Searchable web sites: http://www.hotelguide.ch, a hotel guide, and http://www.bigbook.com, U.S. business listings	Schema description: Each web site exports a single collection of listing objects; HTML page data fields map to attributes of an interface; unique key for a listing provided by web site is key for OID. Method invocation: accessor methods only; generic dispatch. Query operations: attribute projection, equality predicates on attributes, LIKE predicates of the form '%<value>%.
Proprietary database for molecular similarity search	Schema description: A single collection of molecule objects; interface has contains_substructure() and similarity_to() methods to model search capability of engine; molecule l-number is key for OID. Method invocation: stub dispatch. Query operations: attribute and method projection; predicates of the form <attr> <op> <const> and <method> <op> <const>, if <op> is a comparison operator; bind plans if similarity_to() is in bind predicate.
QBIC [16] image server that orders images according to color, texture and shape features	Schema description: Collections of image objects; interface has matches() method to model ordering capability; image file name is key for OID. Method invocation: stub dispatch. Query operations: ordering of image objects by image feature.
Glimpse [15] text search engine that searches for specific patterns in text files	Schema description: Collections of files; interface contains several methods to model text search capability and retrieve relevant text of a file; file name is key for OID. Method invocation: stub dispatch. Query operations: projection of attributes and methods.
Lotus Notes databases: Phone Directory database, Patent Server database	Schema description: Notes database becomes a collection of note objects; interface defined by database Form; note NOTEID is key for Garlic OID. Method invocation: accessor methods only, generic dispatch. Query operations: attribute projection; predicates with logical, comparison and arithmetic operations; LIKE predicates.
Complex Object Wrapper	Schema description: Collections of objects; interface corresponds to interface of objects in database; database OID is key for Garlic OID. Method invocation: stub dispatch. Query operations: attribute projection.

vides a link to the next page of results, so after all of the results on one page are returned to Garlic, the wrapper follows the link and retrieves the next page.

4.5 Wrapper Packaging

In the previous sections, we have described the services that a wrapper provides to the Garlic middleware. The wrapper author's final task is to package these pieces as a complete wrapper. A wrapper may include three kinds of components: *interface files* that contain one or more interface definitions written in GDL, *environment files* that contain name/value pairs to encode repository-specific information for use by the wrapper, and *libraries* that contain dynamically loadable code to implement schema registration, method invocation, and the query interfaces. Libraries are further subdivided as follows: *core* libraries that contain common code shared among several similar repositories, and *implementation* libraries that contain repository-specific implementations of one or more interfaces.

Packaging wrapper code as dynamically loadable libraries that reside in the same address space as Garlic keeps the cost of communicating with a wrapper as low as possible. This is important during query processing, since a given wrapper may be consulted several times during the optimization of a query, and non-trivial data structures are exchanged at each interaction. Very simple repositories can be accessed without crossing address space boundaries, and repositories that are divided into client and server components are easily accommodated by linking their wrapper with the repository's client-side library. This approach encapsulates the choice of a particular client-server protocol (e.g., CORBA-IIOP, ActiveX/DCOM, or ODBC) within the wrapper, allowing Garlic to integrate repositories regardless of the particular protocol(s) they support.

Decomposing wrappers into interface files, libraries, and environment files gives the designer of a wrapper for a particular repository or family of repositories considerable flexibility. For example, our relational wrapper packages generic method dispatch, query planning and query execution code as a sharable core library. For each repository, an interface file describes the objects in the corresponding database. An environment file encodes the name of the database to which the wrapper must connect, the names of the collections exported by the repository and the tables to which they are bound, the correspondence between attributes in interfaces and columns in tables, etc.

Implementation libraries are useful when a wrapper that employs stub dispatch is built for a data source whose schema can evolve over time. As new kinds of objects are added to the repository schema, implementation libraries can be registered with stubs for the new implementations.

5 Current Status

To test the flexibility of our architecture, we have implemented wrappers for a diverse set of 10 data sources. Table 1 describes some of the features of these wrappers. The data models for these sources vary widely, including relational, object-oriented, a simple file system, and a specialized molecular search data model. Likewise, the data sources provide query processing power that ranges from simple scanning to basic predicate application to complex join processing. Wrappers such as the relational wrapper have been fine tuned and are fairly mature. Others, such as the molecular wrapper, are still in a state of evolution.

Based on our experience writing these wrappers, we have identified 3 general categories of wrappers, and provide a base class for each category. We also provide wrapper writers with a library of schema registration tools, query plan construction routines, and other useful routines in order to automate the task of writing a wrapper as much as possible. To test our assertion that wrappers are easy to write, we asked developers outside of the project to write several wrappers listed in the table. For example, a summer student wrote the text and image server wrappers over a period of a few weeks, and a chemist was able to write the molecular database wrapper during a 2-day visit to our lab.

6 Related Work

Presenting a uniform interface to a diverse set of information sources has been the goal of a great deal of previous research, dating back to projects like CCA's Multibase [20]. Surveys of much of this work can be found in [3] [6] [10] [19], and [1] [21] describe actual implementations. In terms of query processing, the architectures of these earlier systems are built around a *lingua franca* for communicating with the underlying sources. These systems assume that any data source, assisted by the translator, can readily execute any query fragment.

OLE DB [2] takes an important step towards integrating heterogeneous data sources by defining a standardized construct, the *rowset*, to represent streams of values obtained from a data source. A simple tabular data source with no querying capability can easily expose its data as a rowset. More powerful data sources can accept commands (either as text or as a data structure) that specify query processing operations peculiar to that data source, and produce rowsets as a result. Thus, although OLE DB does not include a middleware query processing component like Garlic, it does define a protocol by which a middleware component and data sources can interact. This protocol differs from the Garlic wrapper interface in several ways. First, the format of an OLE DB command is defined entirely by the data source which accepts it, whereas Garlic query fragments are expressed in a standard form based on object-extended SQL. Secondly, an OLE DB data source must either accept or reject a command in its entirety, whereas a Garlic wrapper can agree to perform part of a work request and leave any parts it cannot handle to be performed by Garlic.

A different set of techniques for integrating data sources with various levels of query support relies upon an *a priori* declarative specification of query capability for each data source. In the TSIMMIS system [18], specifications of query power are expressed in the Query Description and Translation Language (QDTL) [17]. A QDTL specification for a data source is a context-free grammar for generating supported queries. DISCO [9] builds on the notion of capability records described in the Information Manifold [11] and requires a wrapper writer to describe a data source's capabilities by means of a language based on a set of (relational) logical operators such as `select`, `project`, and `scan`.

The idea of compact declarative specifications of query power is attractive, but there are some practical problems with this approach. First, it is often the case that a data source cannot process a particular query, but can process a *subsuming* query whose answer set includes the answer set of the original query. In general, finding maximal subsuming queries is computationally costly, and choosing the optimal subsuming query may require detailed knowledge of the contents, semantics, and statistics of the repository.

Secondly, in defining a common language to describe all possible repository capabilities, it is difficult to capture the unique restrictions associated with any individual repository. For example, as we noted earlier, relational database systems often place limits on the query string length, the maximum constant value that can appear in a query, etc. Likewise, our web wrapper can handle LIKE predicates, but only if the pattern is of a specific form. The molecular wrapper is sensitive to which attributes and methods appear together in the projection and predicate lists. A language to express these and other repository-specific restrictions would quickly become very cumbersome. Furthermore, in a strictly declarative approach such as DISCO, as new sources are integrated, the language would need to be extended to handle any unanticipated restrictions or capabilities introduced by the new sources.

As we saw in Section 4.3, Garlic forgoes the declarative approach for one in which the knowledge about what a specific repository can and cannot do is encapsulated by the wrapper. Rather than solve the query subsumption problem in general at the Garlic level, we ask wrapper authors to solve the simpler special-case problem for their own repositories. Decisions about how much of a query can be handled by a repository are made by the wrapper at query planning time, taking advantage of repository-specific semantic knowledge. Since our approach is not limited by the expressive power of a query specification language, we can accommodate the idiosyncrasies of almost any data source.

7 Conclusions

In this paper, we have described the wrapper architecture for Garlic, a middleware system designed to provide a unified view of heterogeneous, legacy data sources. Our architecture is flexible enough to accommodate almost any kind of data source. We have developed wrappers for sources that represent a broad spectrum of data models and query capabilities. For sources with specialized query processing capabilities, representing those capabilities as methods has proven to be viable and convenient.

The Garlic wrapper architecture makes the wrapper writer's job relatively simple, and as a result, we have been able to produce wrappers for new data sources in a matter of days or hours instead of weeks or months. Wrapper authoring is especially simple for repositories with limited query power, but even for more powerful repositories, a basic wrapper can be written very quickly. This allows applications to access data from new sources as soon as possible, while subsequent enhancements to the wrapper can transparently improve performance by taking greater advantage of the repository's query capabilities.

Our design also allows the Garlic query optimizer to develop efficient query execution strategies. Our approach does not require a complex language to describe the minute details of the capabilities and restrictions of the underlying data sources. Furthermore, we do not require a wrapper to raise a repository's query processing capabilities to a fixed level, or "dumb down" the query processing interface to the lowest common denominator. Instead, our architecture allows each wrapper to determine on a case-by-case basis how much of a query its repository is capable of handling.

In the future, we will continue to refine the wrapper interfaces. An open research question is to develop a truly satisfactory cost model for a diverse set of data sources. We intend to focus on making the wrapper's job of providing a cost model easier, by providing a basic framework that a wrapper writer can customize for a specific repository. We will also investigate the possibility of introducing QDTL-style templates to allow a wrapper to declare up-front a specification of the expressions it will support. With such information, the Garlic query processor could filter out expressions that a wrapper is unable to handle before the work request is generated. Such a template would be a step toward a hybrid system, combining Garlic's dynamic approach to query planning with the declarative approach of TSIMMIS and DISCO; striking an appropriate balance between the techniques is an interesting research opportunity.

Acknowledgements

We would like to thank the Garlic team members, both past and present, whose hard work and technical contributions made the Garlic project possible. In particular, Laura Haas spent many hours with us working out the details of the query processing interface. We'd also like to thank Mike Carey and Laura Haas for reviewing an initial draft of this paper and providing us with excellent suggestions that improved its presentation and readability.

References

[1] R. Ahmed, et. al., "The Pegasus Heterogeneous Multidatabase System", *IEEE Computer,* 24(12) pp. 19-27, December 1991.

[2] J. Blakely, "Data Access for the Masses Through OLE DB", *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, PQ, Canada, June 1996.

[3] O. Bukhres, and A. Elmagarmid, eds., Object-Oriented Multidatabase Systems, Prentice Hall, publishers, New Jersey, 1996.

[4] M. Carey, et al., "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach", *Proc. IEEE RIDE-DOM*, Taipei, Taiwan, March 1995.

[5] R. Cattell, ed., Object Database Standard: ODMG-93 (Release 1.2), Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[6] A. Elmagarmid and C. Pu, eds., "Special Issues on Heterogeneous Databases", *ACM Comp. Surveys 22(3),* September 1990.

[7] G. Graefe, "Query Evaluation Techniques for Large Data Bases", *ACM Computing Surveys 25(2)*, June 1993.

[8] L. Haas, et. al., "Optimizing Queries Across Diverse Data Sources", *Proc of the 23rd International Conference on Very Large Data Bases,* Athens, Greece, August 1997.

[9] Kapitskaia, O., et. al., "Dealing with Discrepancies in Wrapper Functionality", *INRIA Technical Report RR-3138*, 1997.

[10] W.Kim, Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, Addison-Wesley Publishers, 1995.

[11] A. Levy, et. al., "Querying Heterogeneous Information Sources Using Source Descriptions", *Proc of the 22nd International Conference on Very Large Data Bases,* Bombay, India, September 1996.

[12] G. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives", *Proc. of the ACM SIGMOD Conference on Management of Data,* Chicago, IL, USA, May 1988.

[13] H. Lu and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network", *Proc. 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, August 1985.

[14] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries", in Readings in Database Systems, M. Stonebraker, ed., Morgan-Kaufmann Publishers, San Mateo, CA, 1988.

[15] U. Manber, et. al., http://glimpse.cs.arizona.edu/

[16] W. Niblack, et al., "The QBIC Project: Querying Images By Content Using Color, Texture and Shape", *Proc. SPIE*, San Jose, CA, February1993.

[17] Y. Papakonstantinou, et. al., "A Query Translation Scheme for Rapid Implementation of Wrappers", *Proc. of the Conference on Deductive and Object-Oriented Databases (DOOD)*, 1995.

[18] Y. Papakonstantinou, et. al., "Object Exchange Across Heterogeneous Information Sources", *Data Engineering Conf.,* March 1995.

[19] S. Ram, guest ed., *IEEE Computer Special Issue on Heterogeneous Distributed Database Systems*, 24(12), December 1991.

[20] R. Rosenberg and T. Landers, "An Overview of MULTIBASE", in Distributed Databases, H. Schneider, ed. North-Holland Publishers, New York, NY, 1982.

[21] R. Stout, "EDA/SQL", in Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, ed., pp 649-663, ACM Press, Addison-Wesley Publishers, 1995.

[22] M. Tork Roth, and P. Schwarz, "A Wrapper Architecture for Legacy Data Sources", *IBM Technical Report RJ10077*, 1997, http://www.almaden.ibm.com/cs/garlic/.

Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System [*]

Mary Tork Roth[†] Fatma Özcan[‡] Laura M. Haas[§]

IBM Almaden Research Center, San Jose CA 95120

Abstract

An important issue for federated systems of diverse data sources is optimizing cross-source queries, without building knowledge of individual sources into the optimizer. This paper describes a framework through which a federated system can obtain the necessary cost and cardinality information for optimization. Our framework makes it easy to provide cost information for diverse data sources, requires few changes to a conventional optimizer and is easily extensible to a broad range of sources. We believe our framework for costing is the first to allow accurate cost estimates for diverse sources within the context of a traditional cost-based optimizer.

1 Introduction

Increasingly, companies need to be able to interrelate information from diverse data sources such as document management systems, web sites, image management systems, and domain-specific application systems (e.g., chemical structure stores, CAD/CAM systems) in ways that exploit these systems' special search capabilities. They need applications that not only access multiple sources, but that ask queries over the entire pool of available data as if it were all part of one virtual database. One important issue for such federated systems is how to optimize cross-source queries to ensure that they are processed efficiently. To make good decisions about join strategies, join orders, etc., an optimizer must consider both the capabilities of the data sources and the costs of operations performed by those sources. Standard database optimizers have built-in knowledge of their (sole) store's capabilities and performance characteristics. However, in a world where the optimizer must deal with a great diversity of sources, this detailed, built-in modeling is clearly impractical.

Garlic is a federated system for diverse data sources. Garlic's architecture is typical of many heterogeneous database systems, such as TSIMMIS [PGMW95], DISCO [TRV96], and HERMES [ACPS96]. Garlic is a query processor [HFLP89]; it optimizes and executes queries over diverse data sources posed in an extension of SQL. Data sources are integrated by means of a *wrapper* [RS97]. In [HKWY97, RS97], we described how the optimizer and wrappers cooperate to determine alternative plans for a query, and how the optimizer can select the least cost plan, assuming it has accurate information on the costs of each alternative plan. This paper addresses how wrappers supply information on the costs and cardinalities of their portions of a query plan and describes the framework that we provide to ease that task. This information allows the optimizer to compute the cost of a plan without modifying its cost formulas or building in knowledge of the execution strategies of the external sources. We also show that cost-based optimization *is* necessary in a heterogeneous environment; heuristic approaches that push as much work as possible to the data sources can err dramatically.

Our approach has several advantages. It provides sufficient information for an optimizer to choose good plans, but requires minimal work from wrapper writers. Wrappers for simple sources can provide cost information without writing any code, and wrappers for more complex sources build on the facilities provided to produce more accurate information as needed. Our framework requires few changes to a conventional bottom-up optimizer. As a result, in addition to examining the full space of possible plans, we get the benefits of any advances in optimizer technology for free. The framework is flexible enough to accommodate a broad range of sources easily, and does not assume that sources conform to any particular execution model. We believe that our framework for costing is the first to allow accurate cost estimates for diverse sources within the context of a traditional cost-based optimizer.

The remainder of the paper is structured as follows. Sec-

[*] This work was partially supported by DARPA contract F33615-93-1-1339.

[†] *torkroth@almaden.ibm.com*

[‡] *fatma@cs.umd.edu; current address: Department of Computer Science, University of Maryland; partial funding provided by Army Research Laboratory contract DAAL01-97-K0135.*

[§] *laura@almaden.ibm.com*

**Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.**

tion 2 discusses the traditional approach to costing query plans. In Section 3, we present a framework by which these costing techniques can be extended to a heterogeneous environment. Section 4 shows how a set of four wrappers with diverse capabilities adapt this framework to provide cost information for their data sources. In Section 5, we present experiments that demonstrate the importance of cost information in choosing good plans, the flexibility of our framework, the accuracy it allows, and finally, that it works – the optimizer is able to choose good plans even for complex cross-source queries. Section 6 discusses related work, and in Section 7 we conclude with some thoughts about future directions.

2 Costing in a Traditional Optimizer

In a traditional bottom-up query optimizer [SAC+79], the cost of a query plan is the cumulative cost of the operators in the plan (plan operators, or *POPs*). Since every operator in the plan is the root of a subplan, its cost includes the cost of its input operators. Hence, the cost of a plan is the cost of the topmost operator in the plan. Likewise, the cardinality of a plan operator is derived from the cardinality of its inputs, and the cardinality of the topmost operator represents the cardinality of the query result.

In order to derive the cumulative costs and cardinality estimates for a query plan, three important cost numbers are tracked for each POP: *total cost* (the cost in seconds to execute that operator and get a complete set of results), *re-execution cost* (the cost in seconds to execute the POP a second time), and *cardinality* (the estimated result cardinality of the POP). The difference between total and re-execution cost is the cost of any initialization that may need to occur the first time an operator is executed. For example, the total cost of a POP to scan a temporary collection includes both the cost to populate and scan the collection, but its re-execution cost includes only the scan cost.

The total cost, re-execution cost, and cardinality of a POP are computed using *cost formulas* that model the runtime behavior of the operator. Cost formulas model the details of CPU usage and I/O (and in some systems, messages) as accurately as possible. A special subset of the formulas estimates predicate selectivity.

Cost formulas, of course, have variables that must be instantiated to arrive at a cost. These include the cardinality of the input streams to the operator, and *statistics* about the data to which the operator is being applied. Cardinality of the input streams is either computed using cost formulas for the input operators or is a statistic if the input is a base table. Hence, statistics are at the heart of any cost-based optimizer. Typically, these statistics include information about collections, such as the base cardinality, and about attributes, such as information about the distribution of data values. A traditional optimizer also has statistics about the physical system on which the data is stored, usually captured as a set of constant weights (e.g., CPU speed, disk transfer rate, etc.).

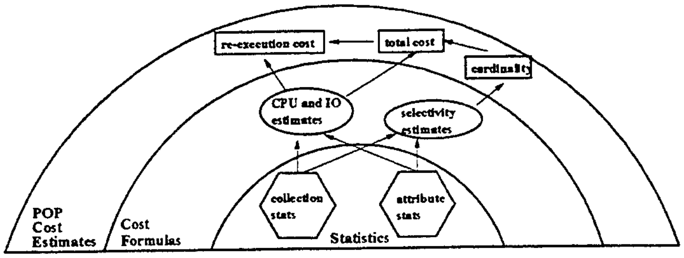


Figure 1: Traditional ost-based optimization

Figure 1 summarizes this flow of information. At the core is a set of statistics that describe the data. At the next layer, these statistics feed cost formulas to compute selectivity estimates, CPU and I/O costs. Finally, in the outer layer, operator costs are computed from the cost formulas, and these operator costs ultimately result in plan costs.

3 Costing Query Plans in a Heterogeneous Environment

This section focuses on the process of costing query plans in a heterogeneous environment. Two significant challenges in adapting a traditional cost-based optimizer to a heterogenous environment are first, to identify *what* additional information is required to cost the portions of a query plan executed by remote sources, and second, *how* to obtain such information. Section 3.1 addresses the *what*, by introducing a framework for wrappers to provide information necessary to extend traditional plan costing to a heterogeneous environment. Section 3.2 addresses the *how*, by describing a default adaptation of the framework and facilities that a wrapper may use to compute cost and cardinality information for its data source.

3.1 A Framework for Costing in a Heterogeneous Environment

While the flow of information from base statistics to plan operator costs described in Section 2 works well in a traditional (relational) environment, it is incomplete for a heterogeneous environment. Given the diversity of data sources involved in a query, it is impossible to build cost formulas into the optimizer to compute the costs of operations performed by those data sources. Furthermore, since the data sources are autonomous, a single strategy cannot be used to scan the base data to gather and store the statistics the optimizer needs to feed its formulas. Clearly, a cost-based optimizer cannot accurately cost plans without cooperation from wrappers. In this section, we describe what information is needed from wrappers to extend cost-based optimization to a heterogeneous environment.

3.1.1 Cost Model

The first challenge for an optimizer in a heterogeneous environment is to integrate the costs of work done by a remote data source into the cost of the query plan. In Garlic, the portions of a query plan executed by data sources are encapsulated as PUSHDOWN POPs. Such POPs show up as leaves of the query plan tree. As a result, total cost, re-execution cost, and result cardinality are all that is needed to integrate the costs of a PUSHDOWN POP into the cost of the query plan.

Fortunately, these three estimates provide an intuitive level of abstraction for wrappers to provide cost information about their plans to the optimizer. On one hand, these estimates give the optimizer enough information to integrate the cost of a PUSHDOWN POP into the cost of the global query plan without having to modify any of its cost formulas or understand anything about the execution strategy of the external data source. On the other hand, wrappers can compute total cost, re-execution cost, and result cardinality in whatever way is appropriate for their sources, without having to comprehend the details of the optimizer's internal cost formulas.

3.1.2 Cost Formulas

Wrappers will need cost formulas to compute their plan costs, and most formulas tailored to the execution models of the built-in operators will typically not be appropriate. On the other hand, some of the optimizer's formulas may be widely applicable. For example, the formula to compute the selectivity of a set of predicates depends on the predicates and attribute value distributions, and not on the execution model. Wrappers should be able to pick from among available cost formulas those that are appropriate for their data sources, and if necessary, develop their own formulas to model the execution strategies of their data sources more accurately.

Additionally, wrappers may need to provide formulas to help the optimizer cost new built-in POPs specific to a heterogeneous environment. For example, traditional query processors often assume that all required attributes can be extracted from a base collection at the same time. In Garlic, wrappers are not required to perform arbitrary projections in their plans. However, they must be able to retrieve any attribute of an object given the object's id. If a wrapper is unable to supply all requested attributes as part of its plan, the optimizer attaches a FETCH operator to retrieve the missing attributes. The retrieval cost may vary greatly between data sources, and even between attributes of the same object, making it impossible to estimate using standard cost formulas. Thus, to allow the optimizer to estimate the cost of this FETCH operator, wrappers are asked to provide a cost formula that captures the *access cost* to retrieve the attributes of its objects.

As another example, wrappers are allowed to export methods that model the non-traditional capabilities of their data sources, and such methods can be invoked by Garlic's query engine. Methods may be extremely complex, and their costs may vary greatly depending on the input arguments. Again, accurately estimating such costs using generic formulas is impossible. Wrappers are asked to provide two formulas to measure a method's costs: *total method cost* (the cost to execute the method once), and *re-execution method cost* (the cost to execute the method a second time). These formulas provide an intuitive level of abstraction for the wrapper, yet give the optimizer enough information to integrate method invocation costs into its operator costs.

3.1.3 Statistics

Both the optimizer and the wrappers need statistics as input to their cost formulas. In a heterogeneous environment, the base data is managed by external data sources, and so it becomes the wrapper's task to gather these statistics. Since wrappers provide the cost estimates for operations performed by their data sources, the optimizer requires only *logical* statistics about the external data. Statistics that describe the physical characteristics of either the data or the hardware of the underlying systems are not necessary or even helpful; unless the optimizer actually models the operations of the data sources, it would not know how to use such statistics.

A traditional optimizer's collection statistics include base cardinality, as well as physical characteristics (such as the number of pages it occupies), which are used to estimate the I/O required to read the collection. In a heterogeneous environment, the optimizer still needs base cardinality statistics to compute cardinality estimates for its operators.

For attributes, optimizers typically keep statistics that can be used to compute predicate selectivity assuming a uniform distribution of values, and some physical statistics such as the average length of the attribute's values. More sophisticated optimizers keep detailed distribution statistics for oft-queried attributes. In a heterogeneous environment, an optimizer still needs some attribute statistics in order to compute accurate cardinality estimates. In Garlic, wrappers are asked to provide uniform distribution statistics (number of distinct values, second highest and second lowest values). They may optionally provide more detailed distribution statistics, and the optimizer will make use of them. Physical statistics such as average column length are not required, although they may be helpful to estimate the cost to operate on the data once it is returned to Garlic. If not provided, the optimizer estimates these costs based on data type.

Not only are these statistics needed for the optimizer's formulas, but wrappers may need them as input to their private cost formulas. In addition, wrappers may introduce *new* statistics that only their cost formulas use. Such statistics may be for collections, attributes, or methods. For example, the cost formulas a wrapper must provide to estimate the total and re-execution costs of its methods are likely to require some information as input. Thus, as with cost formulas, the set of statistics in a heterogeneous environment must be extensible.

To summarize, Figure 2 shows the extended flow of information needed for an optimizer in a heterogeneous environment. White objects represent information that is produced and used by the optimizer. Objects with horizontal lines (e.g., the formula to compute predicate selectivity) are provided by the optimizer and made available to the wrappers. Those with vertical lines are provided by the wrappers, and used by both the optimizer and the wrappers. Statistics and cost formulas shown shaded in gray are introduced by and available only to wrappers. The outer circle shows that wrappers are asked to report the total cost, re-execution cost, and result cardi-

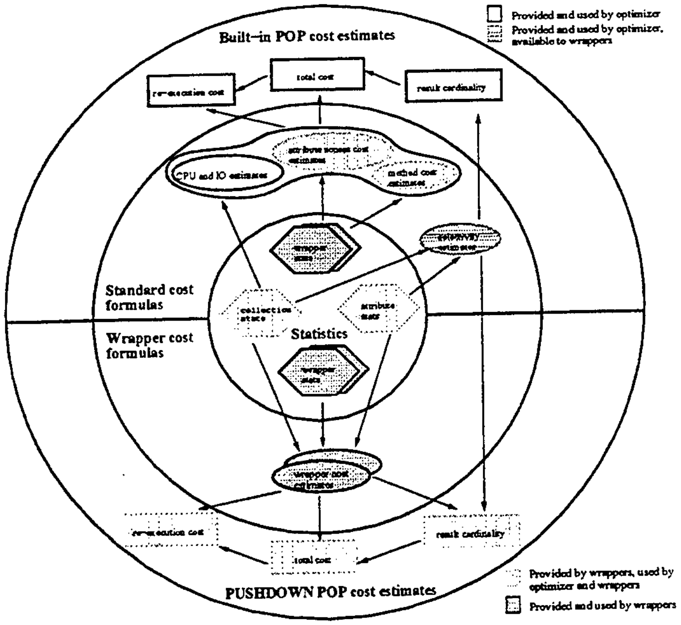


Figure 2: Heterogeneous cost-based optimization

nality for their plans. Armed with this information, the optimizer can combine the costs of PUSHDOWN POPs with the costs of built-in POPs to compute the cost of the query plan. In the next circle, wrappers are asked to provide formulas to compute attribute access costs and method costs. In addition, they can make use of some existing cost formulas, and add new formulas to model the execution strategies of their data sources. Finally, in the inner circle, wrappers are asked to provide the basic statistics about their collections and the attributes of their objects that the optimizer needs as input to its formulas. They may also compute and store statistics that are required by their own formulas.

3.2 Completing the Framework

Figure 2 shows how our framework extends the traditional flow of cost information to include wrapper input at all levels. To make it easy to provide such information (particularly for simple data sources), the framework also provides a default cost model, default cost formulas, and a facility to gather statistics. The framework is completely flexible; wrappers may use any of the defaults provided, or choose to provide their own implementations.

3.2.1 Extending the Cost Model

As described in Section 3.1.1, a wrapper's first job is to report total cost, re-execution cost, and result cardinality for its plans. To make this task as easy as possible, the framework includes a default cost model which wrappers can use to model the execution strategies of their data sources. Wrappers can take advantage of this cost model, or, if it is not sufficient, replace it with a cost model of their own.

The default cost model was designed with simple data sources in mind. We chose this approach for two important reasons. First, simple data sources have very basic capabilities. They can iterate over the objects in their collections, and perhaps apply basic predicates. They do not perform joins or other complex SQL operations. This limited set of capabilities often means that their execution strategy is both straightforward and predictable. These characteristics make it easy to develop a general purpose cost model. Second, an important goal of Garlic is to ensure that writing a wrapper is as easy as possible. If the default cost model is complete enough to model very basic capabilities, then wrapper writers for simple data sources need not provide *any* code for computing costs.

	Cost Model
P1	$plan\_total\_cost = reset\_cost + advance\_cost \times ((result\_cardinality + 1)/BLOCK\_SIZE)$
P2	$plan\_reexecution\_cost = plan\_total\_cost - reset\_cost$
P3	$plan\_result\_cardinality = \prod_{i=1}^{n} BASE\_CARD_i \times applied\_predicates\_selectivity$

Table 1: Default cost model estimates for wrapper plans

The default cost model is anchored around the execution model of a runtime operator. Regardless of whether a runtime operator represents a built-in POP or a PUSHDOWN POP, its work can be divided into two basic tasks[1]: *reset*, which represents the work that is necessary to initialize the operator, and *advance*, which represents the work necessary to retrieve the next result. Thus, the total cost of a POP can be computed as a combination of the reset and advance costs. As shown in Table 1, the default model exploits this observation to compute the total and re-execution costs of a wrapper plan.

(P1), the formula to compute the total cost of a plan, captures the behavior of executing a PUSHDOWN POP once. The operator must be reset once, and advanced to retrieve the complete result set (plus an additional test to determine that all results have been retrieved). *BLOCK_SIZE* represents the number of results that are retrieved at a time. Default formulas to compute reset and advance costs are described in Section 3.2.2 below. The re-execution cost (P2) is computed by factoring out the initialization costs from the total cost estimate. Since PUSHDOWN POPs are leaf POPs of the query plan tree, the result cardinality estimate (P3) is computed by multiplying the cross product of the n collection base cardinalities accessed by the plan by the selectivity of the applied predicates. As described in Section 3.2.3, *BASE_CARD* is the basic collection cardinality statistic, and *applied_predicates_selectivity* can be computed using the standard selectivity formula provided by the optimizer.

3.2.2 Extended Cost Formulas

Our framework provides default implementations of all the cost formulas wrappers need to supply (including those intro-

[1] Our model actually has three tasks; we are omitting discussion of the bind task to simplify exposition. *Bind* represents the work needed to provide the next set of parameter values to a data source, and can be used, for example, to push join predicate evaluation down to a wrapper. However, simple sources typically don't accept bindings, as they cannot handle parameterized queries. Our relational wrapper does accept bindings, and provides cost formulas to calculate their cost.

	Cost formula
F1	$access\_cost(A) = AVG\_ACCESS\_COST_{max} + (n-1) \times OVERHEAD \times AVG\_ACCESS\_COST_{max}$
F2	$method\_total\_cost_i = AVG\_TOTAL\_METH\_COST_i$
F3	$method\_reexecution\_cost_i = AVG\_REEX\_METH\_COST_i$
F4	$reset\_cost = AVG\_RESET\_COST_c$
F5	$advance\_cost = AVG\_ADVANCE\_COST_c$

Table 2: Default cost formulas

duced in Section 3.1.2) as well as those needed by the default cost model of Section 3.2.1. These formulas are summarized in Table 2, and we will describe each formula in greater detail below. They rely on a new set of statistics, and Section 3.2.3 describes how these statistics are computed and stored.

(F1) is the default definition of the attribute access cost formula. A represents a set of n attributes to be retrieved by a FETCH POP. Typically there is a significant charge to retrieve the first attribute, but only an incremental charge to retrieve additional attributes once the first attribute has been retrieved. $AVG\_ACCESS\_COST_i$ is a new attribute statistic that measures the cost to retrieve attribute i, and $AVG\_ACCESS\_COST_{max}$ is the most expensive attribute retrieved by the FETCH. $OVERHEAD$ is a constant multiplier between 0 and 1 that represents the additional cost to retrieve an attribute, assuming that the most expensive attribute in A has already been retrieved. Wrappers may adjust this value as appropriate.

(F2) and (F3) represent the default definitions provided by the framework for the optimizer's method cost formulas. $AVG\_TOTAL\_METH\_COST$ and $AVG\_REEX\_METH\_COST$ are new statistics that represent measures of the average total and re-execution costs to invoke a method. This information is similar to the information standard optimizers keep for user-defined functions [Cor97]. These statistics are extremely simple, and do not, for example, take into account the set of arguments that are passed in to the method. As we will illustrate in Section 4.2, wrappers that use methods to export the nontraditional capabilities of a data source may provide new definitions that match the execution strategy of their underlying data source more accurately, including any dependency on the method's arguments.

Formulas (F4) and (F5) are the default cost formulas used by the default cost model to compute plan costs. For simple wrappers with limited capabilities, computing average times over the range of queries that the data source supports may often be sufficient to obtain accurate cost estimates. The default cost formulas to compute plan costs use this approach. While these formulas are admittedly naive, in Section 5.2 we show that they work remarkably well for the simple data sources in our experiments. Since simple wrappers typically do not perform joins, the reset and advance costs are computed to be the average reset and advance costs of the single collection c accessed by the plan. AVG_RESET_COST and $AVG\_ADVANCE\_COST$ are new collection statistics that represent measures of the average time spent initializating and retrieving the results for queries executed against a collection.

If these default cost formulas are not sufficient for a particular data source (and they won't be for more capable sources), a wrapper writer may provide formulas that more accurately reflect the execution strategy of the data source. In fact, our framework for providing cost formulas is completely extensible; wrappers may use the optimizer's predicate selectivity formulas, any of the default formulas used by the default cost model, or add their own formulas. Wrapper-specific formulas can feed the formulas that compute operator costs and cardinalities, and their implementations can make use of the base statistics, and any statistics wrappers choose to introduce.

Category	Statistic	Query template
Collection	BASE_CARD	select count(*) from collection
	AVG_RESET_COST*, AVG_ADVANCE_COST*	select c.OID from collection c
Attribute	NUM_DISTINCT_VALUES	select count(distinct c.attribute) from collection c
	2ND_HIGH_VALUE	select c.attribute from collection c order by 1 desc
	2ND_LOW_VALUE	select c.attribute from collection c order by 1 asc
	AVG_ACCESS_COST*	select c.attribute from collection c
Method	AVG_TOTAL_METH_COST*, AVG_REEX_METH_COST*	select c.method(args) from collection c

Table 3: Statistics generated by update_statistics facility

3.2.3 Gathering Statistics

As described in Section 3.1.3, both the standard cost formulas and wrapper-provided cost formulas are fueled by statistics about the base data. Garlic provides a generic *update_statistics* facility that wrappers can use to gather and store the necessary statistics. The update_statistics facility includes a set of routines that execute a workload of queries against the data managed by a wrapper, and uses the results of these queries to compute various statistics. Wrappers may use this facility "as-is", tailor the query workload to gain a more representative view of the data source's capabilities and data, or augment the facility with their own routines to compute the standard set of statistics and their own statistics.

Table 3 describes the default query workloads that are used to compute various statistics. From the table, it should be clear how the standard statistics are computed. However, the calculations for the newly introduced statistics (marked with an asterisk) bear further description. For collections, the default cost model described in Section 3.2.1 relies on statistics that measure the average time to initialize and advance a wrapper operator. These measures are derived by executing a workload of single-collection queries defined by the wrapper writer (or DBA) that characterizes the wrapper's capabilities. For example, for a simple wrapper, the workload may contain a single simple "select c.OID from collection c" query, executed multiple times. Running averages of the time spent in reset and advance

Wrapper	Code	Cost model	Cost formulas	Statistics
ObjectStore	0	default	default	default
Lotus Notes	0	default	default	default
QBIC	700	default	replaces method cost, reset, advance formulas	added method statistics
Relational	400	default	replaces reset, advance formulas	added collection statistics

Table 4: Wrapper adaptations of framework

of the wrapper's runtime operator are computed for this workload of queries, and those measures are stored as the *AVG_RESET_COST* and *AVG_ADVANCE_COST* statistics. Note that these times include network costs, so the plan cost formulas do not have to consider network costs explicitly.

To compute the new attribute statistic *AVG_ACCESS_COST* (used by the default cost formula to compute attribute access cost), a single query which projects the attribute is executed, and the optimizer is forced to choose a plan that includes a `FETCH` operator to retrieve the attribute. This query is executed multiple times, and a running average of the time spent retrieving the attribute is computed, including network costs.

For the new method statistics, a workload of queries which invoke the method with representative sets of arguments (supplied by the wrapper writer) is executed multiple times. Running averages are computed to track the average time (including network costs) to execute the method both the first time, and multiple subsequent times. These averages are stored as the *AVG_TOTAL_METH_COST* and *AVG_REEX_METH_COST* statistics, respectively.

4 Wrapper Adaptations of the Framework

Figure 2 shows how our framework enables wrapper input in each concentric circle. In this section, we describe how a set of wrappers have adapted this framework to report cost information about their data sources to the optimizer. These wrappers represent a broad spectrum of capabilities, from the very limited capabilities of our complex object repository wrapper to the very powerful capabilities of our wrapper for relational databases. Table 4 summarizes how these different wrappers adapted the framework to their data sources, as well as the number of lines of code involved in the effort.

4.1 Simple Data Sources

We use ObjectStore as a repository to store complex objects, which Garlic clients can use to bind together existing objects from other data sources. This wrapper is intentionally simple, as we want to be able to replace the underlying data source without much effort. This wrapper will only generate plans that return the objects in the collections it manages, i.e., it will only produce plans for the simple query "`select c.OID from collection c`". The optimizer must append Garlic POPs to the ObjectStore wrapper's plans to fetch any required attributes and apply any relevant predicates. Because of its simplicity, the wrapper uses the default cost model to compute its plan costs and cardinality estimates, and uses the update_statistics facility "as-is" to compute and store the statistics required to fuel the default formulas, as well as those used by the optimizer to cost the appended Garlic POPs. We will see in Section 5.2 that the default model is indeed well-suited to this very basic wrapper.

We also implemented a more capable wrapper for Lotus Notes databases. It can project an arbitrary set of attributes and apply combinations of predicates that contain logical, comparison and arithmetic operators. It cannot perform joins. We have observed that the execution strategy for Lotus Notes is fairly predictable. For any given query with a set of attributes to project and set of predicates to apply, Lotus will retrieve each object from the collection, apply the predicates, and return the requested set of attributes from those objects that survive the predicates.

We intended to demonstrate with this wrapper that only a few modifications by the wrapper were needed to tailor the default cost model to a more capable wrapper. However, as we will show in Section 5.2, we discovered that although the wrapper for Lotus Notes is much more capable than the ObjectStore wrapper, the behavior of its underlying data source is predictable enough that the simple default cost model is still suitable. The wrapper writer was only required to tailor the workload of queries used to generate the *AVG_RESET_COST* and *AVG_ADVANCE_COST* collection statistics so that they more accurately represented the data source's capabilities. We used a simple workload of queries; techniques described in [ZL98] might do a better job of choosing the appropriate sample queries.

4.2 Data Sources with Interesting Capabilities

QBIC [N$^+$93] is an image server that manages collections of images. These images may be retrieved and ordered according to features such as average color, color histogram, shape, texture, etc. We have built a wrapper for QBIC that models the average color and color histogram feature searches as methods on image objects. Each method takes a sample image as an argument, and returns a "score" that indicates how well the image object on which the method was invoked matched the sample image; the lower the score, the better the match. These methods may be applied to individual image objects via the Garlic method invocation mechanism. In addition, the QBIC wrapper will produce plans that apply these methods to all image objects in a collection, and return the objects ordered from best match to worst match. It can also produce plans that return the image objects in an arbitrary order. It does not apply predicates, project arbitrary sets of attributes, or perform joins.

Both the average color feature searches and color histogram feature searches are executed in a two-step process by the QBIC image server. In the first step, an appropriate 'color value' is computed for the sample image. In the second step, this value is compared to corresponding pre-computed

Feature	*sample_eval_cost*	*comparison_cost*
Average color	*AVG_COLOR_SLOPE* × *sample_size* + *AVG_COLOR_INTERCEPT*	*AVG_COLOR_COMPARE*
Color histogram	*AVG_HISTOGRAM_EVAL*	*HISTOGRAM_SLOPE* × (*number_of_colors*) + *HISTOGRAM_INTERCEPT*

Table 5: QBIC wrapper cost formulas

values for the images in the scope of the search. In our implementation, the scope is a single object if the feature is being computed via method invocation, or all objects in the collection if the feature is being computed via a QBIC query plan. For each image in the collection, the relationship between the sample image's color value and the image's color value determines the score for that image. Hence, the cost of both feature searches can be computed using the following general formula:

$$search\_cost = sample\_eval\_cost + m \times comparison\_cost$$

In this formula, *sample_eval_cost* represents the cost to compute the color value of the sample image, *comparison_cost* represents the cost to compare that value to a collection image object's corresponding value, and *m* is the number of images in the scope of the search.

In an average color feature search, the color value represents the average color value of the image. The execution time of an average color feature search is dominated by the first step and depends upon the x-y dimensions of the sample image; the larger the image, the more time it takes to compute its average color value. However, the comparison time is relatively constant per image object. The first entry in Table 5 shows the formulas the wrapper uses to estimate the cost of an average color feature search. *AVG_COLOR_SLOPE*, *AVG_COLOR_INTERCEPT*, and *AVG_COLOR_COMPARE* are statistics the wrapper gathers and stores using the update_statistics facility. The wrapper uses curve fitting techniques and a workload of queries with different sizes for the sample image to compute both the *AVG_COLOR_SLOPE* and *AVG_COLOR_INTERCEPT* statistics. *AVG_COLOR_COMPARE* represents the average time to compare an image, and an estimate for it is derived using the same workload of queries.

In a color histogram feature search, the color value represents a histogram distribution of the colors in an image. In this case, the execution time is dominated by the second step. For the typical case in which the number of colors is less than six, QBIC employs an algorithm in which the execution time of the first step is relatively constant, and the execution time of the second step is linear in the number of colors in the sample image. The second entry of Table 5 shows the formulas the wrapper uses to compute the cost of color histogram searches. *AVG_HISTOGRAM_EVAL*, *HISTOGRAM_SLOPE*, and *HISTOGRAM_INTERCEPT* are new statistics, and *number_of_colors* represents the number of colors in the sample image. Again, the wrapper uses curve fitting and a workload of queries with different numbers of colors for the sample image to compute *HISTOGRAM_SLOPE* and *HISTOGRAM_INTERCEPT*. It uses the same workload of queries to compute an average cost for *AVG_HISTOGRAM_EVAL*.

The wrapper uses these formulas to provide both method cost estimates to the optimizer and to compute the costs of its own plans. The effort to provide these formulas and compute the necessary statistics was about 700 lines of code.

4.3 Sophisticated Data Sources

Our relational wrapper is a "high-end wrapper"; it exposes as much of the sophisticated query processing capabilities of a relational database as possible. Clearly, the default formulas are not sufficient for this wrapper. Vast amounts of legacy data are stored in relational databases, and we expect performance to be critically important. The time invested in implementing a more complete cost model and cost formulas for a relational query processor is well-worth the effort.

However, decades of research in query optimization show that modelling the costs of a relational query processor is not a simple task, and creating such a detailed model is not within the scope of this paper. We believe that an important first step is to implement a set of formulas that provide reasonable ball-park cost estimates, as such estimates may be sufficient for the optimizer to make good choices in many cases. With this goal in mind, for the relational wrapper, we chose to use the default cost model, and implement a very simple set of formulas to compute the reset and advance costs that use an oversimplified view of relational query processing:

$$reset\_cost = prep\_cost$$
$$advance\_cost = execution\_cost + fetch\_cost$$

In these formulas, *prep_cost* represents the cost to compile the relational query, *execution_cost* represents the cost to execute the query, and *fetch_cost* represents the cost to fetch the results. At a high level, *prep_cost* and *execution_cost* depend on the number of collections involved in the query, and *fetch_cost* depends on the number of results. The relational wrapper used curve fitting techniques and the update_statistics facility to compute and store cost coefficients for these formulas. The total number of lines to implement this was less than 400.

While this is an admittedly naive implementation, the estimates produced by this formula are more accurate than those from the default model, and provide the optimizer with accurate enough information to make the right plan choices in many cases. However, we do not claim that our implementation is sufficient for the general case. We believe many of the techniques applied in [DKS92] and the approaches of more recent work of [ZL98] could be adapted to work within the context of the relational wrapper, and present an interesting area of research to pursue.

5 Experiments and Results

In this section, we describe the results of experiments that show that the information provided by wrappers through our

	Query	Pushdown join time (secs)	Garlic join time (secs)	Card
Q1	**select** p.id, p1.id **from** professor p, professor p1 **where** p.id < p1.id **and** p.status = p1.status **and** p.aysalary > 115000 **and** p1.aysalary > 115000	369.47	1550.52	605401
Q2	**select** p.id, p1.id **from** professor p, professor p1 **where** p.id < p1.id **and** p.status < p1.status **and** p.aysalary > 115000 **and** p1.aysalary > 115000	6332.14	1766.11	2783677

Table 6: A comparison of execution times for 2 join queries

framework is critical for the optimizer to choose quality execution plans. Without wrapper input, the optimizer can (and will) choose bad plans. However, with wrapper input, the optimizer is able to accurately estimate the cost of plans. As with any traditional cost-based optimizer, it may not always choose the optimal plan. However, for most cases, it chooses a good plan and avoids bad plans.

We adapted the schema and data from the BUCKY benchmark [C+97] to a scenario suitable for a federated system. We used only the relational schema, distributed it across a number of sources, and added to it a collection of images representing department buildings. We developed our own set of test queries that focus on showing how the optimizer performs when data is distributed among a diverse set of data sources. The test data is distributed among four data sources: an IBM DB2 Universal Database (UDB) relational database, a Lotus Notes version 4.5 database, a QBIC image server, and an ObjectStore version 4.0 object database. For the experiments, the query engine executed on one machine, the UDB database, QBIC image server, and ObjectStore database all resided on a second server machine, and the Notes server resided on a third machine. All were connected via a high-speed network. When an execution plan included a join, we limited the optimizer's choices to nested loop join and pushdown join. This did not affect performance, and allowed us to illustrate the tradeoffs in executing a join in Garlic or at a data source without having to consider countless alternative plans. It should be noted that Garlic is an experimental prototype, and as such, the Garlic execution engine is slower than most commercial relational database product engines. However, it is significantly faster than Notes. Hence, we believe our test environment is representative of a real world environment in which some sources are slower and some faster than the middleware, and hence, is a fair testbed for our study.

5.1 The Need for Wrapper Input

This first set of experiments addresses the need for cost-based optimization in an extensible federated database system. It has been suggested [ACPS96, EDNO97, LOG93, ONK+96, SBM95] that heuristics that push as much work as possible to the data sources are sufficient. Consider the two queries defined in Table 6. (Q1) finds all pairs of similarly ranked professors that make more than $115,000 a year. (Q2) finds, for all professors that make at least $115,000 a year, the set of professors of a lower rank that also make at least $115,000 a year. The professor collection is managed by the relational wrapper. There are two obvious plans to execute these queries: push both the join and the predicate evaluation down to the UDB wrapper, or push the predicate evaluation down to the wrapper but perform the join in Garlic. Table 6 also shows the result cardinality and execution times for these 2 plans. In (Q1), the equi-join predicate on status restricts the amount of data retrieved from the data source, so the pushdown join is a better plan. However, in (Q2), the join predicates actually increase the amount of data retrieved from the data source, so it is faster to execute the join in Garlic. These queries represent two points in a query family that ranges from an equi-join (p.status = p1.status) to a cross product (no predicate on status). At some point in this query family, there is a *crossover point* at which it no longer makes sense to push the join down. The crossover point depends on different factors, such as the amount of data, the distribution of data values, the performance of both query engines, network costs, etc. Cost-based optimizers use such information to compare plan alternatives to identify where such crossover points exist, while heuristic approaches can only guess.

ID	Department predicate	Professor predicate	Cardinality
Q3	dno < 1	id = 101	0
Q4	dno < 51	id = 101	50
Q5	dno < 101	id = 101	100
Q6	dno < 151	id = 101	150
Q7	dno < 201	id = 101	200
Q8		id < 102	250
Q9		id < 103	500
Q10		id < 105	1000
Q11		id < 107	1500
Q12		id < 109	2000

Table 7: UDB professorxdepartment predicates

5.1.1 Working without wrapper input

The previous example motivated the need for cost-based optimization in a federated system by showing that pushing down as much work as possible to the data sources is not always a winning strategy. In this experiment, we show that accurate information is crucial for a cost-based optimizer to identify crossover points. For this set of experiments, we chose a family of queries over the UDB department and professor collections. To control result cardinality, we used a cross product with local predicates (shown in Table 7) on each table.

To predict plan costs accurately, a cost-based optimizer depends heavily on the availability and accuracy of statistics. If statistics are not available, the optimizer uses default values for these parameters. Without accurate information, the optimizer will sometimes choose a good plan, and sometimes it will not. In our environment, in the absence of wrapper input,

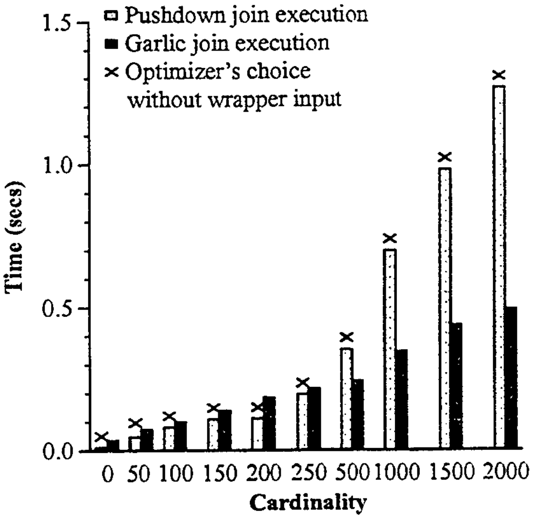


Figure 3: Optimizer choices without wrapper input

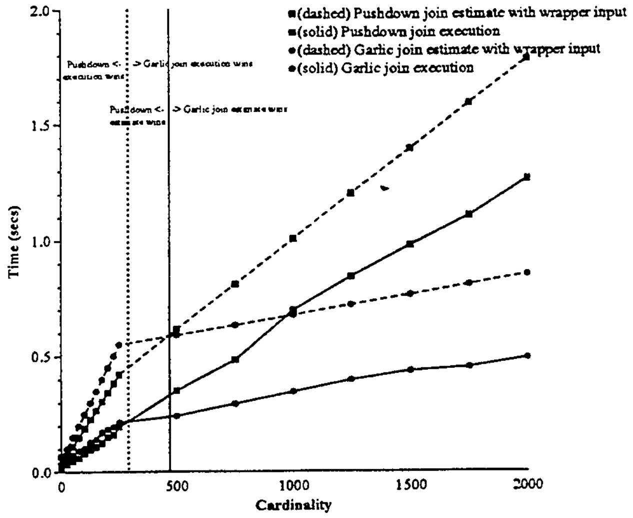


Figure 4: Optimizer estimates with statistics

the optimizer's parameters have been tuned to favor pushing as much work down to the data sources as possible.

For the set of queries in Table 7, the crossover point at which it makes sense to execute the join in Garlic occurs between queries (Q8) and (Q9), or when the result cardinality is between 250 and 500. Figure 3 shows the execution times for executing these queries with both the pushdown join and Garlic join plans. For each query, an x marks the plan that was chosen by the optimizer. Since the optimizer does not have the benefit of wrapper input, it relies on its defaults, and favors the pushdown join plan in all cases. With only default values, the cardinalities of the base collections look the same, and all local predicates (e.g., d.dno < 101 or p.id < 102) have the same selectivity estimates. Without more accurate information, the optimizer cannot easily discriminate between plans.

5.1.2 Working with wrapper input

Consider the same set of queries, only this time with input from the UDB wrapper, using the cost model and formulas described in Section 4.3. Figure 4 shows both the optimizer's estimates and the execution times for both the pushdown and Garlic join plans. The graph shows that while the optimizer's estimates differ by 10% to 45% from the actual execution costs, the wrapper input allows the optimizer to compare the *relative* cost of the two plans. Keep in mind that the cost formulas implemented by the UDB wrapper are fairly naive; if the wrapper writer invested more effort in implementing cost formulas reflecting the execution strategies of UDB, the optimizer's estimates would be more accurate.

Now instead of favoring the pushdown plan in all cases, the optimizer recognizes a crossover point in which it makes sense to execute the join in Garlic. The vertical dotted line on the graph shows the actual crossover point. The vertical solid line on the graph shows the optimizer's estimate of the crossover point. The area between the two lines represents the range in which the optimizer may make the wrong choice. Since we didn't have a data point in this area of the graph, we ran further experiments to identify the range more accurately. These experiments used a few more predicates to allow us to control the result cardinality more precisely. We found that the execution crossover point is at cardinality = 251, and the optimizer identifies the crossover point in the 278-298 range. Thus, the range in which the optimizer will make the wrong choice is between 251 and at most 298. In this narrow range, the execution times of the plans are so close that the wrong plan choice is not significant.

5.2 Adaptability of the Framework

In the previous section, we showed that wrapper input is critical for the optimizer to choose good plans. In this section, we show that our framework makes it easy for wrappers to provide accurate input. We look at 3 wrappers in particular: ObjectStore, Notes, and QBIC.

5.2.1 Wrappers that Use the Default Cost Model

As described in Section 4.1, the ObjectStore wrapper is our most basic wrapper and uses the default cost model without modification. [ROH99] shows the optimizer's estimates and actual execution times for a set of queries that exercise the wrapper's capabilities. The experiments show that the defaults are well suited for the ObjectStore wrapper; the optimizer's estimates differ from the actual execution time by no more than 10%.

Recall that although Notes is a more capable wrapper, the Notes wrapper also uses the default cost model, formulas, and statistics. Again, [ROH99] shows that for a set of queries that exercise the wrapper's capabilities, the optimizer's estimates are "in the ballpark", ranging from a 13% to 40% difference from the actual execution time. For the experiments with more complicated queries, the optimizer's estimates are off by more than 30%. Further analysis showed that a significant percentage of this difference can be attributed to result cardinality underestimates, which were off by 21% for both of these queries. Such inaccuracies are not unusual for cost-

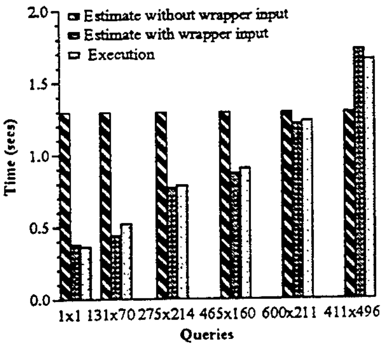


Figure 5: QBIC avg color query plan

based optimizers, and are the result of imperfect cost formulas and deviations in the data from the distribution assumptions. To make up the difference between the estimate and the actual execution time that cannot be attributed to inaccurate result cardinality estimates, the wrapper writer could provide formulas that model the predicate application strategy of Lotus Notes more accurately. However, we do not believe such effort is necessary. Analysis to be presented in section 5.3 shows that even for this more capable wrapper, the default cost formulas provide estimates that are close enough for the optimizer to choose good plans in most instances.

5.2.2 Wrappers with Interesting Capabilities

For data sources with unusual capabilities, such as QBIC, the default model is not sufficient. As described in Section 4.2, the execution time for an average color search depends on the size of the sample image. Figure 5 shows optimizer estimates and actual execution times for a family of average color queries with increasingly larger predicate images. The x-axis shows the size of the sample image. The first bar for each query represents the optimizer's cost estimate without wrapper input, the second bar shows the optimizer's cost estimate with wrapper input, and the third bar shows the actual execution time.

Without wrapper input, the optimizer has no knowledge of how much an average color search costs, nor is it aware that the cost depends on the size of the sample image. Thus, it must rely on default estimates, which can in no way approximate the real cost of the search or the plan. However, with wrapper input, the optimizer's estimates do reflect the dependency on the image predicate size, and its estimates are extremely accurate, with most being within 4% of the actual cost. An analysis of color histogram queries yields similar results. As we will see in Section 5.3, such input from wrappers with unusual capabilities is crucial for the optimizer to choose good plans when data from that source is joined with data from other sources.

5.3 Cross-Repository Optimization

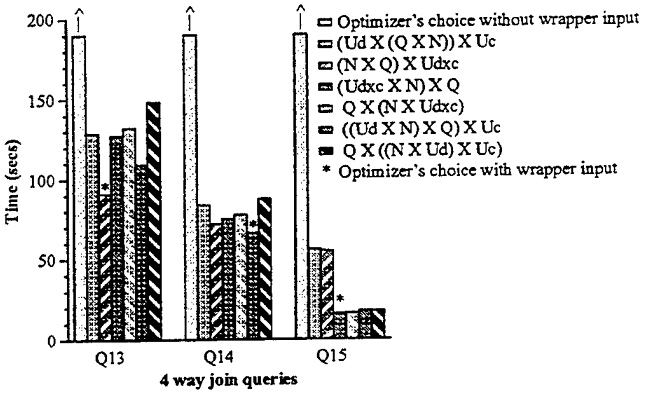


Figure 6: 4-way cross-repository join queries

Query template
select i.OID, i.avg_color('767x589_image.gif'), i.avg_color('1x1_image.gif') **from** images i, notes_departments n_d, udb_course u_c, udb_department u_d **where** n_d.building = i.image_file_name **and** u_c.dno = u_d.OID **and** u_d.dno = n_d.dno

Table 8: 4-way join query template

Our final experiment shows that our framework provides sufficient information for the optimizer to choose good plans for complex queries. For this experiment, we used the query template given in Table 8 to generate a query family. The query template is a 4-way join between the department and course collections managed by the UDB wrapper, the Notes department collection, and the QBIC image collection. To generate the family, we added predicates on the UDB department collection and the UDB course collection that control the cardinality of the results. These predicates and the result cardinalities are shown in Table 9. The queries also contain 2 average color image searches, one of which is for a 1x1 image (cheap), while the other is for a 767x598 image (expensive).

The number of possible plans for executing this query family is over 200. However, a large number of these plans are clearly bad choices, as they would require computing large cross-products. We enumerated and forced the execution of the 20 most promising plans, including the ones the optimizer itself selected. In any plan, the optimizer is forced to push one average color search down and evaluate the other by method invocation because the QBIC wrapper returns plans that execute only one search at a time.

Figure 6 shows the execution time of 7 plans for each query. The first bar represents the plan the optimizer chose without statistics or wrapper input. The other 6 bars are representative plans from the set that we analyzed. The plans are

ID	Predicates	Card
Q13	u_d.budget < 10000000 and u_c.cno < 102	456
Q14	u_d.budget < 6000000 and u_c.cno < 102	258
Q15	u_d.budget < 2000000 and u_c.cno = 102	23

Table 9: 4-way join query predicates

denoted by the order in which the joins are evaluated. A collection is identified by the first character of the wrapper that manages it. The UDB collections are further marked by the first character of each collection. An upper case X indicates that the join was done in Garlic, and a lower case x indicates the join was pushed down to the UDB wrapper. A * over a bar indicates that the optimizer, working with wrapper input, chose the corresponding plan.

For all three queries, the optimizer picked the best plan of the alternatives we studied, and, we believe, of all possible plans. Note that this may not happen in general; the purpose of a cost-based optimizer is not to choose the optimal plan for every query, but to consistently choose good plans and avoid bad ones.

The graph once again reinforces the assertion that wrapper input is crucial for the optimizer to choose the right plan. Without wrapper input, the optimizer chose the same plan for all three queries, which was to push the join between the UDB collections down to the UDB wrapper, join the result of that with the Notes department collection, and join that result with the image collection. Without information from the QBIC wrapper about the relative costs of the two image searches, it arbitrarily picked one of them to push down, and the other to perform via method invocation. In this case, the optimizer made a bad choice, pushing the cheap search of the 1x1 image down to the QBIC wrapper, and executing the expensive search via method invocation on the objects that survive the join predicates. This plan is a bad choice for all three queries, with execution times well over 1000 seconds.

When the optimizer was given input from the QBIC wrapper about the relative cost of the two average color searches, it chose correctly to push the expensive search down to the QBIC wrapper and perform the cheap search via method invocation. This is true for all plans we looked at for all queries, and brings the execution times for all of our sample plans to under 200 seconds.

This experiment also shows that pushing down as much work as possible to the data sources does not always lead to the best plan. For (Q13) and (Q15), the best plan did in fact include pushing the join between the UDB collections down to the UDB wrapper. However, for (Q14), the best plan actually split these two collections, and joined UDB department with Notes department as soon as possible. In this plan, the predicate on the UDB department collection (u_d.budget < 6000000) restricted the number of UDB department tuples by 50%. Joining this collection with the Notes department collection first also reduced the number of tuples that needed to be joined with the image collection by 50%. For (Q13), the UDB department predicate (u_d.budget < 10000000) was not as restrictive. In this case, it would have only reduced the number of tuples that needed to be joined with the image collection by 9%, which was not a significant enough savings to make this alternative attractive. Instead, it was better to group UDB department and UDB course together and push the join down to the UDB wrapper.

For (Q15), the UDB department predicate is even more restrictive, filtering out over 90% of the tuples. In this case, it is a good idea to use it to filter out both the Notes department tuples and UDB course tuples as soon as possible. Thus, the two best plans push the join between the UDB collections down to the wrapper, and immediately join the result with Notes. The two worst plans failed to take advantage of this. Plan 2 in the figure arranged these collections out of order, and plan 3 joined the entire Notes department collection with QBIC image before the join with the UDB collections.

These experiments show that cost-based optimization is indeed critical to choose quality execution plans in a heterogeneous environment. Using our framework, wrappers can provide enough information for the optimizer to cost wrapper plans with a sufficient degree of accuracy. By combining such cost information with standard cost formulas for built-in operators, traditional costing techniques are easily extended to cost complex cross-source queries in a heterogeneous environment.

6 Related Work

As federated systems have gained in popularity, researchers have given greater attention to the problem of optimizing queries over diverse sources. Relevant work in this area includes work on multidatabase query optimization [LOG93, DSD95, SBM95, EDNO97, ONK+96] and early work on heterogeneous optimization [Day85, SC94], both of which focus on approaches to reduce the flow of data for cross-source queries, and not on estimation of costs. More recent approaches [PGH96, LRO96] describe various methods to represent source capabilities. Optimizing queries with foreign functions[CS93, HS93] is related, but these papers have focused on optimization algorithms, and again, not on estimating costs. [UFA98] describes orthogonal work to incorporate cost-based query optimization into query scrambling.

Work on frameworks for providing cost information and on developing cost models for data sources is, of course, highly relevant. OLE DB [Bla96] defines a protocol by which federated systems can interact with external data sources, but it does not address cross-source query optimization, and presumes a common execution model. The most complete framework for providing cost information to date is Informix's DataBlades [Cor97] architecture. DataBlades integrates individual tables, rather than data sources, and the optimizer computes the cost of an external scan using formulas that assume the same execution model as for built-in scans.

Various approaches have been proposed to develop cost models for external data sources. These approaches can be grouped into four categories: calibration [DKS92, GST96], regression [ZL98], caching [ACPS96], and hybrid techniques [NGT98]. The calibration and regression approaches typically assume a common execution model for their sources (which doesn't work for heterogeneous federations), but may be useful in developing wrapper cost models for particular sources. Both [ACPS96] and [NGT98] deal with diverse data sources, but neither approach employs standard dynamic programming optimization techniques.

7 Conclusion

We have demonstrated the need for cost-based optimization in federated systems of diverse data sources, and we presented a complete yet simple framework that extends the benefits of a traditional cost-based optimizer to such a federated system. Our approach requires only minor changes to traditional cost-based optimization techniques, allowing us to easily take advantage of advances in optimization technology. Our framework provides enough information to the optimizer for it to make good plan choices, and yet, it is easy for wrappers to adapt. In the future, we intend to continue testing our framework on a broad range of data sources. We would like to add templates to support classes of data sources that share a common execution model, and test our framework for how well it handles object-relational features such as path expressions and nested sets.

8 Acknowledgements

We thank Peter Haas, Donald Kossmann, Mike Carey, Eugene Shekita, Peter Schwarz, Jim Hafner, Ioana Ursu and Bart Niswonger for their help in preparing this paper, and V.S. Subrahmanian for his support.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[Bla96] J. Blakely. Data access for the masses through ole db. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[C^{+}97] M. Carey et al. The bucky object-relational benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–146, Tucson, Arizona, US, May 1997.

[Cor97] Informix Corporation. Guide to the virtual table interface. Manual, 1997.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.

[Day85] U. Dayal. Query processing in a multidatabase system. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 81–108. Springer, 1985.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Canada, 1992.

[DSD95] W. Du, M.-C. Shan, and U. Dayal. Reducing multidatabase query response time by tree balancing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 293–303, San Jose, CA, USA, May 1995.

[EDNO97] C. Evrendilek, A. Dogac, S. Nural, and F. Ozcan. Multidatabase query optimization. *Distributed and Parallel Databases*, 5(1):77–114, 1997.

[GST96] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 378–389, Bombay, India, September 1996.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J .Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[LOG93] H. Lu, B.C. Ooi, and C.H. Goh. Multidatabase query optimization: Issues and solutions. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 137–143, 1993.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[N^{+}93] W. Niblack et al. The QBIC project: Querying images by content using color, texture and shape. In *Proc. SPIE*, San Jose, CA, USA, February 1993.

[NGT98] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Conf. on Data Engineering*, Orlando, Florida, USA, 1998.

[ONK^{+}96] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization on a distributed object management platform. In *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, pages 117–124, Rockville, MD, USA, 1996.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami, Fl, USA, December 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[ROH99] M. Tork Roth, F. Ozcan, and L. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. *IBM Technical Report RJ10141*, 1999.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[SAC^{+}79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SBM95] S. Salza, G. Barone, and T. Morzy. Distributed query optimization in loosely coupled multidatabase systems. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, pages 40–53, Prague, Czech Republic, January 1995.

[SC94] P. Scheuermann and E. I. Chong. Role-based query processing in multidatabase systems. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pages 95–108, Cambridge, England, March 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. of the Intl. Conf on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, 1996.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, USA, June 1998.

[ZL98] Q. Zhu and P. Larson. Solving local cost estimation problem for global query optimization in multidatabase systems. *Distributed and Parallel Databases*, 6:1–51, 1998.

Loading a Cache with Query Results

Laura M. Haas
IBM Almaden

Donald Kossmann
University of Passau

Ioana Ursu
IBM Almaden

Abstract

Data intensive applications today usually run in either a client-server or a middleware environment. In either case, they must efficiently handle both database queries, which process large numbers of data objects, and application logic, which involves fine-grained object accesses (e.g., method calls). We propose a wholistic approach to speeding up such applications: we load the cache of a system with relevant objects as a by-product of query processing. This can potentially improve the performance of the application, by eliminating the need to fault in objects. However, it can also increase the cost of queries by forcing them to handle more data, thus potentially reducing the performance of the application. In this paper, we examine both heuristic and cost-based strategies for deciding what to cache, and when to do so. We show how these strategies can be integrated into the query optimizer of an existing system, and how the caching architecture is affected. We present the results of experiments using the Garlic database middleware system; the experiments demonstrate the usefulness of loading a cache with query results and illustrate the tradeoffs between the cost-based and heuristic optimization methods.

1 Introduction

Data intensive applications today usually run in either a middleware or client-server environment. Examples of middleware systems include business application, e-commerce or database middleware systems, while CAD and CAE systems are typically client-server. In either case, they must efficiently handle both database queries, which process large numbers of data objects, and application logic, with its fine-grained object accesses (e.g., method calls). In both architectures, application logic and query processing may be co-resident, and take place on a processor other than that on which the data resides. It is increasingly likely that some or all of the data will be on remote and/or nontraditional data sources that are expensive to access, such as web sources or specialized application systems.

Sophisticated optimization techniques reduce query processing times in these environments, while caching is used to reduce the cost of the application logic by avoiding unnecessary requests to the data sources. Applications often ask queries to identify objects of interest and then manipulate the result objects. Though it is now possible to do chunks of application logic in the query processor, applications still do much of the work themselves. Some applications require user interaction; others desire greater portability and ease of installation (e.g., big business applications such as Baan IV, Peoplesoft 7.5, or SAP R/3). In traditional systems, query processing and caching decisions are made in isolation. While this provides acceptable performance for these systems, it is a disaster for applications using data from the Internet. This query-and-manipulate pattern means that traditional systems access the data twice: once while processing the query, and then again, on the first method call, to retrieve and cache the object. If data is on the Internet, this will be prohibitively expensive. In some cases, the data source may not even be able to look up individual objects; hence this extra round trip is impossible.

In this paper we propose to load the cache with relevant objects as a by-product of the execution of a query. With this technique it is possible to get orders of magnitude improvements for applications that involve both queries *and* methods over expensive-to-access data. However, a naive implementation can do more harm than good. An application today can manually cache query results by explicitly selecting all the data for the object in the query itself. However, this may increase the cost of queries dramatically by forcing them to handle more data. For complex queries this effect may be large enough to more than offset the benefit. Therefore, the decisions of what to cache and when during query execution to do so should be made by the query optimizer in a cost-based manner.

The remainder of this paper is organized as follows. In Section 2, we elaborate on the motivation for our work, and discuss the caching of objects in our environment. While loading a cache with query results is essential when data is expensive or difficult to access, our ap-

**Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.**

proach can also be used to speed up applications in traditional two- or three-tier architectures as described above. For ease of exposition we will talk about "middleware" as the site of query processing and caching in the following sections. In a two-tier system, these activities would take place in the client. Section 3 presents alternative ways to extend an optimizer to generate query execution plans that load a cache with query results. We describe two simple heuristics, as well as a more sophisticated cost-based approach. Section 4 discusses our implementation of caching in the Garlic database middleware system, and Section 5 contains the results of performance experiments that demonstrate the need to load a cache with query results and show the tradeoffs of the three alternative ways of extending the query processor. Section 6 discusses related work, and Section 7 concludes the paper.

2 Caching in Middleware

2.1 A Motivating Example

To see why loading the cache with query results is useful, consider this (generic) piece of application code:

```
foreach o, o2, o5 in
        (select r.oid, s2.oid, s5.oid
        from R r, S1 s1, S2 s2, ...
        where ...)
      { ...o.method(o2, o5); ... }
```

The query in this example is used to select relevant objects from the database. After further analysis and/or user interaction, the method carries out operations on these objects. The query can be arbitrarily complex, involving joins, subqueries, aggregation, etc. The method will involve accesses to certain fields of the object o and possibly to other objects (o_2, o_5) as well. *r.oid* refers to the object identifier of an object of collection R; this identifier is used to invoke methods on the object and to access fields of the object. Such a code fragment could be found in many applications. For example, an inventory control program might select all products for which supplies were low (and their suppliers and existing orders). After calculating an amount to order (perhaps with user input), it might invoke a method to order the product.

In a traditional middleware system this code fragment is carried out as follows:

1. the query processor tries to find the best (i.e., lowest cost) plan to execute the query.

2. the query processor executes the query, retrieving the object ids requested.

3. an interpreter executes the method, using the object ids to retrieve any data needed. To speed up the execution of methods that repeatedly access the same objects, the interpreter uses caching. Requests to access objects already in the cache can be processed by the interpreter without accessing the underlying data source(s), and a request to access an object not found in the cache would result in *faulting in* that object.

The key observation is that query processing does not affect caching in traditional systems: if the relevant objects of R are not cached prior to the execution of the query, these objects will not be cached as a by-product of query execution and they will have to be faulted in at the beginning of each method invocation. In an environment in which data access is slow, this can be extremely expensive – just as expensive, in fact, as processing the query. Loading the cache with query results avoids this extra cost of faulting in objects by copying the R objects into the cache while the query is executed; that is, it seizes the opportunity to copy the R objects into the cache at a moment at which the objects must be accessed and processed to execute the query anyway.

2.2 Caching Objects

Our goal is to decrease the overall execution time of applications, such as those described above, that use queries to identify the objects on which they will operate (i.e., on which they will invoke methods). There are many possible ways to accomplish this goal. In this paper, we focus on speeding up method execution, by essentially "pre-caching" the objects that methods will need. This pre-caching is possible in our environment, first, because, in executing the query, the query processor has to touch the needed objects anyway, and second, because in the architectures we consider, some portion of the query processing is done at the same site as that at which the methods are executed. Hence, the query processor has the opportunity to copy appropriate objects into a cache, for the methods to use.

Obviously, it will only be beneficial to cache objects that are subsequently accessed by the application program. Ideally, one would carry out a data flow analysis of the application program [ASU89] in order to determine which objects of the query result are potentially accessed. Unfortunately, such data flow analyses are impossible in many cases due to the separation of application logic and query processing – and interactive applications are totally unpredictable. Thus some heuristic approach to identifying the relevant objects is needed. It is likely that the objects whose oids are returned as part of the query result (i.e., objects of collections whose oid columns are part of the query's SELECT clause) are going to be accessed by the application program subsequently (why else would the query ask for oids?)[1]. We refer to such collections as *candidate collections*; these collections are candidates because objects of these collections are likely to be accessed. However, they are not guaranteed to be cached, as it might nevertheless not be cost-effective.

[1] Alternatively, we could assume that the application programmer gives hints that indicate which collections should be cached.

In the applications we are considering, queries and methods run in the same transaction. Hence we are only interested in intra-transaction caching in this paper, and cache consistency is not an issue. Our approach is particularly attractive in environments that do not support inter-transaction caching because transactions start with no relevant objects in the cache. Issues of locking and concurrency control are orthogonal to loading a cache with query results. In a middleware environment, it is often undesirable or impossible to lock data in the sources for the duration of the transaction. Under such circumstances, our approach may cause an application to produce different output; but, in some sense, this output can be seen as better output because our approach guarantees that the methods see the same state of an object as the query.

In this paper, we assume that the granularity of caching is an entire object. (We discuss how an object is defined in Section 4.) To cache the object, the whole object must be present. One may argue that we should only copy those fields of objects that are retrieved as part of the query anyway. However, state-of-the-art caches cache in the granularity of whole objects (e.g., the cache of SAP R/3 [KKM98]). This is necessary for pointer swizzling [Mos92, KK95] and to organize the cache efficiently (i.e., avoid a per attribute overhead in the cache). One may also argue that the granularity of caching and data transfer should be a group of objects or a *page*; caching and data transfer in such a granularity, however, is not possible in systems like Garlic.

A consequence of this approach is that caching during query execution is not free. It introduces additional cost, as no attribute of an object may be projected out before the object is cached, even if the attribute is not needed to compute the query result. This has two implications. First, objects should only be cached if the expected benefit (overall application speedup due to faster execution of methods) outweighs the cost in query execution time. Second, the point in the query execution at which objects are cached will affect the cost and benefit. If caching occurs too early, irrelevant objects may be cached, and might even flood the cache, squeezing out more relevant objects. If caching occurs too late, the intermediate results of query processing will be larger due to the need to preserve whole objects for caching. Consider, for instance, a query that involves a join between R and S and asks for the *oid* of the R objects that qualify: joining only the *oid* column of R with S is cheaper than joining the *whole* R (i.e., *oid* and all other columns) with S, especially if the *oid* column of R fits into main memory and the *whole* R does not. Because caching impacts the size of intermediate results, it should also impact join ordering; for instance, joins that filter out many objects of candidate collections should perhaps be carried out early in a query plan if caching is enabled. Hence, the best way of executing the query may be different depending on whether we are caching objects.

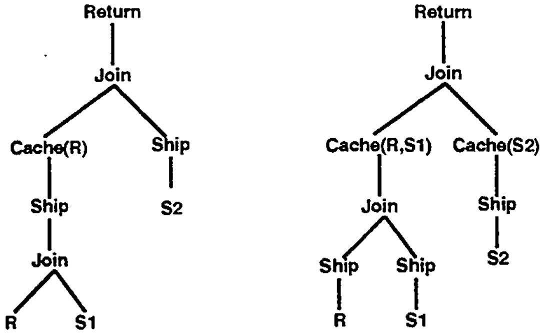


Figure 1: Example Cache Enhanced Query Plans

In summary, our goal is to speed up the execution of methods by caching the objects they need (as indicated by the select list of a query) during execution of that query. The granularity of the cache is an object, and caching objects during query execution incurs costs that can affect the choice of query execution plan. As a result, we will allow the query optimizer to decide what objects to cache, and when.

3 Caching During Queries

In this section, we describe ways to extend the query processor of a middleware system in order to generate plans which cache relevant objects. We introduce a new *Cache* operator which the query optimizer can use to indicate where in a plan objects of a particular collection should be cached. A *Cache* operator copies objects from its input stream into the cache and projects out columns of the input stream which are not needed to produce the query results. A *Cache* operator takes two parameters, one that specifies which objects of the input stream should be copied into the cache, and one that specifies which columns should be "passed through" to the next operator (not projected out). The plans shown in Figure 1 could be produced by the enhanced query optimizer. The *Cache* operator of the first plan copies objects of collection R; the first *Cache* operator of the second plan copies objects of R and S_1 while the second copies S_2 objects. A plan may contain several *Cache* operators if the objects of more than one collection are to be cached; however, it makes no sense to have two *Cache* operators for the same collection in a plan. The *Return* operators pass the query results to the application program. The *Ship* operators pass intermediate query results from a data source to the middleware; since *Cache* operators are always executed by the middleware, all *Cache* operators must be placed somewhere above a *Ship* operator and below the final *Return* operator.

In order to generate such plans, the query optimizer must decide (1) for which collections involved in a query to include *Cache* operators in a query plan, and (2) where to place *Cache* operators in a query plan. We present three approaches. The first two are heuristics which serve as baselines for our study. The third approach is cost-based cache operator placement: this approach is likely to make better decisions (i.e., produce better plans), but increases the cost of query optimization.

3.1 Cache Operators at the Top of Query Plans

The first approach makes the two cache operator placement decisions in the following heuristic way: (1) generate a *Cache* operator for every candidate collection, and (2) place all *Cache* operators high in a query plan. This approach corresponds to what an application could do manually, and is based on the principle that *all* relevant objects (objects which are part of the query result and belong to candidate collections) should be cached during the query and *no* irrelevant objects (those not part of the query result) should be cached. In detail, this approach works as follows:

1. rewrite the `SELECT` clause of a query, replacing all occurances of `oid` by `*`.
2. optimize the rewritten query in the conventional way.
3. include *Cache* operators for the collections whose *oid* columns are requested in the `SELECT` clause of the query, and place those *Cache* operators at the top of the query plan generated in Step 2 (i.e., just below the *Return* operator); remember that *Cache* operators carry out projections so that the right columns for the original query are returned.
4. push down *Cache* operators through *non-reductive* operators. A non-reductive operator is an operator that does not filter out any objects. Examples are *Sort* operators and certain functional joins for which integrity constraints guarantee that all objects satisfy the join predicate(s) (see [CK97] for a formal definition of non-reductive operators).

The push-down of *Cache* operators through non-reductive operators (Step 4) reduces the cost of executing the query and at the same time obeys the principle that only relevant objects are copied into the cache. Suppose, as an example, that a *Cache* operator is pushed below a *Sort*: the cost of the *Sort* is reduced because the *Sort* operator works on *thin* tuples, because the *Cache* operators project out all the columns that were added as part of the rewriting in Step 1. At the same time, no irrelevant objects are copied into the cache because the *Sort* does not filter out any objects.

While pushing down *Cache* operators through non-reductive operators is certainly an improvement, this "caching at the top" approach clearly does not always produce good cache-enhanced plans. Because *Cache* operators impact the size of intermediate results, the placement of *Cache* operators should also impact join ordering; however, the heuristic ignores this interdependency. Furthermore, *Cache* operators high in a plan force lower operators to handle *thick* tuples with high extra cost. The heuristic basically assumes that the extra cost incurred by plans with *Cache* operators is always outweighed by the benefit of these *Cache* operators for (future) method invocations – an assumption which is not always valid, even when data accesses are expensive (Section 5.4).

3.2 Cache Operators at the Bottom of Query Plans

The second approach, "caching at the bottom", makes the following cache operator placement decisions: (1) generate a *Cache* operator for every candidate collection, and (2) place all *Cache* operators low in a query plan. Like the "caching at the top" heuristic, the "caching at the bottom" heuristic assumes that the benefits of *Cache* operators for candidate collections always outweigh the cost incurred by the presence of *Cache* operators. However, the "caching at the bottom" heuristic places *Cache* operators low in query plans, following the principle that columns which are only needed for caching and not to evaluate the query itself should be projected out as early as possible. Thus, the "caching at the bottom" approach affects the cost of other query operators (i.e., joins, group bys, etc.) as little as possible, but it might copy objects into the cache that are not part of the query result and which would be filtered by these other query operators.

In detail, the "caching at the bottom" approach works as follows:

1. optimize the original query in the conventional way.
2. for each leaf node of the resulting plan, if the operator accesses a candidate collection, expand the list of attributes returned to include all the attributes of the objects.
3. place a *Cache* operator for that collection above each such leaf operator.
4. pull up *Cache* operators that sit below pipelining operators (e.g., filters or nested-loop joins).

Cache operator pull-up in the "caching at the bottom" approach is analogous to *Cache* operator push-down in the "caching at the top" approach. Push-down heuristics reduce the cost of a query without increasing the number of *false cache insertions* (adding objects to the cache that do not participate in the query result, hence will not be used later). Pull-up heuristics reduce the number of *false cache insertions* without increasing the cost of a query. Consider as an example a *Cache* operator that sits below a pipeline operator which filters out some of its input tuples. Moving the *Cache* operator above that pipeline operator will reduce the number of objects copied into the cache, without increasing the cost of the pipeline operator because the cost of a pipeline operator does not depend on the width of the tuples it processes.

3.3 Cost-based Cache Operator Placement

It should be clear from the previous two subsections that there is a fundamental tradeoff between "high" and "low" *Cache* operators: the higher a *Cache* operator, the lower the number of *false cache insertions*, and the higher the number of other query operators that sit below the *Cache* operator and operate at increased cost because they must process *thick* tuples. The "caching at the top" and "caching at the bottom" heuristics attack this tradeoff in simple ways; obviously, there are situations in which either one or even both approaches do not find the best place to position a *Cache* operator in a query plan.

In this section, we show how a query processor can make *Cache* operator placement decisions in a cost-based manner. The approach is based on the following extensions:

1. extend the enumerator to enumerate alternative plans with *Cache* operators
2. estimate the cost and potential benefit of *Cache* operators to determine the best plan; the cost models for other query operators (e.g., joins, etc.) need not be changed
3. extend the pruning condition of the optimizer to eliminate sub-optimal plans as early as possible

We describe these three extensions in more detail in the following subsections.

3.3.1 Enumeration of Plans with *Cache* Operators

The implementation of the cost-based placement strategy is integrated with the planning phase of the optimizer. We discuss the necessary changes in the context of a bottom-up dynamic programming optimizer [SAC+79]. Optimizers of this sort generate query plans in three phases. In the first phase, they generate plans for single collection accesses. In the next phase, they generate plans for joins. They first enumerate the two-way joins, using the plans built in the first phase as input. Likewise, they then plan three-way joins, using the plans previously built (for single collections and two-way joins), and so on, until a plan for the entire join is generated. The final phase then completes the plan by adding operators for aggregation, ordering, unions, etc. Each plan has a set of *plan properties* that track what work has been done by that plan. In particular, they record what collections have been accessed, what predicates applied, and what attributes are available, as well as an estimated cost and cardinality for the plan[2]. Each operator added to a plan modifies the properties of that plan to record what it has done. At the end of each round of joins, as well as at the end of each phase, the optimizer *prunes* the set of generated plans, finding plans which have done the same work (have the same properties) and eliminating all but the cheapest.

Plan 1: Index Scan - A_{thick}
Plan 2: Index Scan - A_{thin}
Plan 3: Relation Scan - A_{thick}
Plan 4: Relation Scan - A_{thin}
Plan 5: Cache(A) - Ship - Index Scan - A_{thick}
Plan 6: Cache(A) - Ship - Relation Scan - A_{thick}

Figure 2: Plans for Accessing Table A

Only a few changes need to be made to an existing optimizer to allow it to generate plans with *Cache* operators. First, we have to define what a *Cache* operator does to a plan's properties. *Cache* projects out (i.e., does not pass on to higher operators) unneeded attributes, so it changes the attribute property. It also will affect the cost, as discussed in Section 3.3.2 below. Next, the first and second phases must be modified to generate alternative plans with *Cache* operators. In modern dynamic programming optimizers [Loh88, HKWY97], this corresponds to adding one *rule* to each of those phases. In the access phase, in addition to the normal (*thin*) plans for a collection, which select out just the attributes needed for the query, the new rule will also generate plans for getting all the attributes of the objects in the collection (*thick plans*), if the collection is one of those whose *oid* column is selected by the query (i.e., a candidate collection). In addition, the rule will generate extra plans which consist of a *Cache* (and *Ship*) operator above each of the thick plans. Figure 2 shows the six plans that would be generated in phase one of enumeration if the collection access could be done by either scanning the collection or by scanning an index. If *thick* and *thin* coincide (i.e., all columns of *A* are needed to produce the query result, regardless of caching), only four plans would be enumerated, as Plans 1 and 3 would be identical to 2 and 4, respectively.

Similarly, in the join planning phase, the enumerator must consider possible caching plans in addition to normal join plans. Since there will be a thick plan for each candidate collection, we will automatically get joins with thick result objects. On top of these, we add appropriate *Cache* operators during each round of joining. We can consider caching any subset of *available* candidate collections in a given plan, where *available* means that the plan's properties indicate that that collection has been accessed, that no other *Cache* operator for that collection is present in the plan, and that the full objects are present (it's a thick plan for that collection). This, of course, can cause an exponential explosion in the number of plans that must be considered. For example, Figure 3 shows four basic join plans and five caching plans for a two table join query; actually, even more plans are possible taking into account that more than one join method is applicable and that *Ship* operators can be placed before or after the joins. In Section 3.3.3, we discuss how ag-

[2] There are several other properties that are tracked; we only list the most relevant for this paper.

Plan 1: Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 2: Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thick}
Plan 3: Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thin}
Plan 4: Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thin}
Plan 5: Cache(A) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 6: Cache(B) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 7: Cache(A,B) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 8: Cache(B) - Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thick}
Plan 9: Cache(A) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thin}

Figure 3: Plans Generated for $A \bowtie B$
A, B are candidate collections

gressive pruning can help control this explosion.

3.3.2 Cost/Benefit Calculation of *Cache* Operators

Since *Cache* operators can only be applied on whole objects, their presence increases the cost of underlying operators (because these underlying operators must work on more data). Further, since *Cache* operators project out the columns not needed for the query result, their properties (other than cost) are the same as a simple (non-caching) thin plan. For example, Plans 2, 4, 5 and 6 in Figure 2 have the same properties, excluding cost. Plans with *Cache* operators have done more work to get to the same point; they can survive, therefore, only if the *Cache* operators have a negative cost. At the beginning of optimization, a potential *benefit* is computed for each collection to be cached. The *cost* of a *Cache* operator is defined as the *actual cost* to materialize its input stream *minus* the estimated *benefit*, or savings, from not faulting in objects in future method invocations. The actual cost of the *Cache* operator is proportional to the cardinality of the input plan, and represents the time to copy objects into the cache, and do the project to form the output stream.

The benefit is considerably trickier to estimate. Fortunately, a reasonably detailed model is possible, and is sufficient for choosing good plans. To compute the benefit of a collection, we need to know how many distinct objects of the collection will be part of the query result. For simplicity, we will refer to this number as the *output* of the collection for this query. We assume that the application will invoke methods on a certain fraction F (e.g. 80 %) of the objects in the query result. The benefit, B, is proportional to the output, O: $B = k \times F \times O$, where k represents the time to fault in the object[3]. k, F, and the output of a collection are constant for a given query; they do not depend on the plan for the query, or when (or if) caching occurs. Thus, the benefit can be computed before planning begins. For complete accuracy, B should include a factor f_1 representing the fraction of the relevant objects not already in the cache; however, the overhead to estimate f_1 is not justified given the accuracy we can achieve for other parts of the formula, so we ignore this factor and assign F a lower value accordingly.

The tricky part is how to estimate the output. One approach is to let the optimizer do it. For this alternative, to find the output of a collection R, the optimizer is asked to plan a modified version of the original query, such that the original select list is replaced by "distinct R.oid". The result cardinality of this query is the required output. Note that since the plan for this modified query is unimportant, the optimizer can use any greedy or heuristic approach it wants to reduce optimization time, as long as it does use its cardinality estimation formulas. However, this approach is still likely to be expensive, especially for large queries in which multiple collections are candidates for caching, as the optimizer will be called once per candidate collection, and then again to plan the actual query. Nor is the result guaranteed to be accurate; it will be only as good as the optimizer's cardinality estimates.

Instead, we devised a simple algorithm for estimating output [HKU99]. This approach has much less overhead and estimates the output of a collection with accuracy close to that of the optimizer for queries where the join predicates are independent. The algorithm takes a query as input, and returns an estimate of the output of each candidate collection for the query. The algorithm essentially emulates the optimizer's cardinality computations, but without building plans. It starts by estimating the effect of applying local predicates to the base collections, using the optimizer formulas. It then heuristically chooses an inner for each join and "applies" the join predicate to the inner's output. The output of a collection is taken to be the minimum value among its initial cardinality, its output after applying the most selective local predicate (if any) and its output after applying the most selective join predicate (if any). The algorithm seems to provide a good compromise between accuracy and overhead, though it needs tuning for joins over composite keys.

3.3.3 Pruning of Plans with *Cache* Operators

At the end of each phase of planning, and at the end of each round of joins, the optimizer examines the plans that have been generated, and "prunes" (i.e., throws away) those which are at least as expensive as some other plan that has equivalent or more general properties. Thin plans are less general (because they make available fewer attributes) than thick ones; hence, although thick plans

[3] k depends on the data source and object. [ROH98] describes how an optimizer can assess the value of this parameter.

are typically more expensive, they will not naturally be pruned in favor of thin plans.

This is good, in terms of ensuring that all possible caching plans are examined. However, as described in Section 3.3.1, it also leads to an exponential explosion in the number of plans. Fortunately, since the *Cache* operator only passes through those attributes needed for the query, it creates thin plans (or at least, thinner plans) that compete with each other. For example, in Figure 2, of the six plans generated for accessing collection A in the first phase of optimization, at most two will survive: one thick plan and one thin plan (if it is cheaper than the thick one). The thin survivor could either be a caching plan (e.g., Plan 6) or an original thin plan (e.g., Plan 2). In the join phase, the maximum number of plans that survives each round is 2^n, where n is the number of candidate collections in this round. So in Figure 3, four plans could survive: one in which both A and B are thick, one in which both are thin, one in which A is thick and B thin, and one in which B is thick and A thin (for example, the survivors might be Plans 1, 2, 6 and 7).

However, under certain conditions we can safely prune a thick plan in favor of a thin – and the sooner we eliminate such plans the better for optimization times. In particular, we can prune the thick plan for a candidate collection A if:

$$Cost_{A_{thin}} \leq Cost_{A_{thick}} + Cost_{A_{CacheBest}} - Benefit$$

where $Cost_{A_{CacheBest}}$ is the minimum actual cost incurred to cache a collection and corresponds to the case where the *Cache* operator sits directly above that join that results in the minimum number of output tuples from the collection. It can be computed before optimization, during the output calculations described in Section 3.3.2. The condition basically says that if we assume the minimal possible cost for caching A (lowest actual cost less constant benefit), and that is still more than the cost of a thin plan for A, then there is no point in keeping the thick plan, as caching A is not a good idea.

3.4 Other Strategies and Variants

In this section, we presented three alternative ways to generate plans with *Cache* operators. These three approaches mark cornerstones in the space of possible strategies for integrating *Cache* operator placement into a query processor. The first two approaches are simple strategies that always place *Cache* operators either at the top or at the bottom of query plans. Neither approach causes much overhead during query optimization, but they are likely to make sub-optimal decisions in many cases. The third approach is a full-fledged, cost-based approach for determining cache operator placement. This approach can be the cause of significant additional overhead during query optimization, but is likely to make good decisions.

We can imagine many approaches that make better decisions than the "caching at the top" and "caching at the bottom" heuristics at the expense of additional overhead, or approaches that are cheaper than "cost-based caching" at the risk of making poor decisions in some cases. We describe here just a few variants:

cost-based Cache operator pushdown: rather than push *Cache* operators down through *non-reductive* query operators only, this variant would push a *Cache* operator down through another operator if the result would be a lower cost plan, using the cost model and cost/benefit calculations for *Cache* operators of Section 3.3.2.

cost-based Cache operator pull-up: *Cache* operator pull-up can also be carried out during post-processing of plans in a cost-based manner, instead of pulling *Cache* operators up only through pipeline operators.

flood-sensitive Cache operator elimination: The "caching at the bottom" variant can be extended in such a way that *Cache* operators that would flood the cache because they are applied to too many objects (according to the cardinality estimates of the optimizer) are eliminated from the plan.

rigorous pruning in cost-based approach: There are several possible variants of the "cost-based caching" approach which more aggressively prune plans, even when it may not be wholly "safe" to do so (in other words, they may discard plans that could be the basis of winning plans later on). These variants reduce the cost of query optimization considerably, at the expense of perhaps missing good plans. For example, one aggressive variant might generalize the pruning condition of Section 3.3.3, and always keep at most one of the alternative plans at the end of the round. A somewhat gentler variant might keep two plans at the end of each round of plan generation: a "pure" thick plan, that is, a plan in which all attributes of all candidate collections of the plan are present, and a "pure" thin plan, that is, a plan in which no attributes not necessary for the original query are present.

4 Implementation Details

We implemented all three cache operator placement strategies described in the previous section and integrated them into the Garlic database middleware system. In this section, we describe the major design choices we made in our implementation.

4.1 Double Caching Architecture

Figure 4 shows the overall design of the cache manager and query execution engine. Our implementation involves a double caching scheme. There is a *primary cache* used by the application, while *Cache* operators load objects into a *secondary cache* during query execution. From the secondary cache these objects are copied into the primary cache when they are first accessed by a method. *Resident object tables* (ROT) in both the primary and secondary cache are used to quickly find an

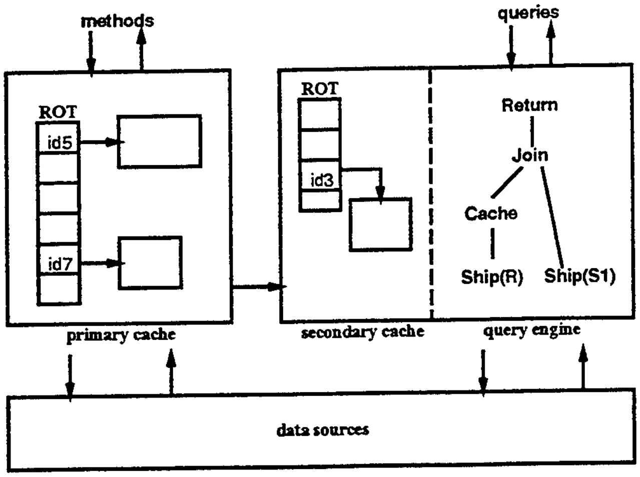


Figure 4: Double Caching Architecture

object in the cache. *Cache* operators only copy objects into the secondary cache that are not present in either the primary or the secondary cache. Thus, they waste as little main memory for double caching as possible and avoid copying objects into the secondary cache multiple times if the input stream of the *Cache* operator contains duplicates. During method invocations, an object is faulted into the primary cache from the data sources if it is not found in the primary or the secondary cache, just as in a traditional middleware system.

The double caching scheme shown in Figure 4 has two important advantages. First, copying objects into a secondary cache, rather than directly into the primary cache, prevents the primary cache from being flooded with query results, thus displacing frequently used objects. Consider, for example, a case in which the query optimizer estimates that the *Cache* operator copies, say, 100 objects; but in fact, the optimizer errs because of outdated statistics and the *Cache* operator would in fact copy millions of objects into the cache. The double caching scheme makes it possible to control and limit the impact of *Cache* operators. Second, the overhead of copying objects into the cache as a by-product of query execution can be reduced in such a double caching scheme. In the primary cache, objects are managed and replaced in the granularity of objects—this is reasonable because individual objects are faulted in and replaced in the primary cache during method invocations. The secondary cache, on the other hand, is organized in chunks; that is, when a *Cache* operator begins execution it will allocate space for, say, 1000 objects in the secondary cache, knowing that it is likely to copy many objects. In other words, the double caching scheme makes it possible to efficiently *bulkload* the cache with relevant objects.

However, the double caching scheme also has some disadvantages: (1) it incurs additional computational overhead in order to copy objects from the secondary cache into the primary cache when the objects are needed; (2) it does waste main memory because after an object has been copied from the secondary into the primary cache, it is cached twice; (3) it requires some (albeit little) tuning effort—this is the flip side of the coin which provides better control over the impact of *Cache* operators. In our experience, the advantages of the double caching scheme outweigh these disadvantages, but, in general, the tradeoffs strongly depend on the kind of application being processed by the middleware system.

4.2 Caching in Middleware for Diverse Sources

Garlic has been designed with an emphasis on handling diverse sources of information, especially sources that do not have traditional database capabilities, though they may offer interesting search and data manipulation capabilities of their own. Loading the middleware cache with query results is particularly attractive for systems like Garlic. First, communicating with some sources may be expensive in Garlic; almost any Web source, for example, will have a highly variable and typically long response time. In such situations, the benefit of *Cache* operators is particularly high (i.e., parameter k is large). Second, some sources are unable to just produce an object given its oid; that is, they do not support the faulting in of objects. Applications that operate on data stored in such data sources *must* load relevant objects as a by-product of query execution; otherwise, such applications simply cannot be executed.[4]

Loading the middleware cache with query results also raises several challenges in this environment. Diverse sources have diverse data. It may not always be practical to cache an entire object. For example, an object may have large and awkward attributes that should only be brought to the middleware if they are really needed. Alternatively, it may be desirable to cache values that are actually computed by methods of a data source because these values are frequently referenced by application programs. So, a flexible notion of "object" is needed. Garlic provides some flexibility in defining objects. Garlic communicates with sources by way of wrappers [RS97]. A wrapper writer must understand the data of a source and describe it in terms of objects. The description can indicate for each attribute (and method) of an object whether it should be part of the cached representation of the object. Garlic has access to this description during query processing, and can use it to decide what attributes and/or methods to include in a thick plan. Ideally, however, we would cache *application objects* which could include data from several collections, possibly from different data sources, and let programmers define such *application objects* for each application program individually. At present we have no mechanism to cache such user-defined application objects, but caching the underlying objects serves the same purpose, by bringing the data needed to construct the application

[4] In such situations, our cost-based approach must be extended to make sure that the winning plan contains a *Cache* operator.

Collection	Base cardinality	Data source
course	12,000	UDB
department	250	UDB
coursesection	50,000	UDB
professor	25,000	UDB
student	50,000	UDB
kids	116,759	UDB
NotesCourses	12,000	Notes
NotesDepartments	250	Notes
WWWPeople	25,000	WWW

Table 1: Test Data Sources and Object Collections

Query	Data sources	Output cardinality
select c.oid from course c where c.deptno<11	UDB	500
select c.oid from NotesCourses c where c.course_dept<11	Notes	500
select p.oid from WWWPeople where p.WWWcategory = 'professor' and p.WWWname like 'professorName15%'	WWW	500

Table 2: Benchmark Queries for Experiment 1

object to the middleware server.

5 Experiments and Results

This section presents the results of experiments that demonstrate the utility (and even, the necessity) of loading a cache with query results by studying the overall running times of applications that involve queries and methods. Next, we look at how query planning time is affected by the three *Cache* operator placement strategies. Finally, we compare the quality of plans produced by the three approaches. We begin with a description of the experimental environment.

5.1 Experimental Environment

The experiments were carried out in the context of the Garlic project, using the double caching architecture described in Section 4.1. For our experiments, we adapted the relational schema and data from the BUCKY benchmark [CDN+97] to a scenario suitable for a federated system. The test data is distributed among three data sources: an IBM DB2 Universal Database (UDB), a Lotus Notes version 4.5 database, and a World Wide Web (WWW) source. The WWW source is populated with data from UDB at the time of query execution using IBM's Net.Data product. The data collections, base cardinalities, and distribution among data sources are shown in Table 1. The Garlic middleware and the UDB and WWW databases run on separate IBM RS/6000 workstations under AIX; the Notes database resides on a PC running Windows NT. All machines are connected by Ethernet. In all experiments, the middleware cache is initially empty.

5.2 Experiment 1: The Value of Caching

The first set of experiments shows the importance of caching in general, and of our *enhanced caching* (loading the cache with query results) in particular. We measured the running times of three simple application programs that initiate the execution of a query and invoke two methods on each object of the query result. The queries used in the three application programs are given in Table 2; they are simple one-table queries against the UDB, Lotus Notes, and WWW databases. For these simple queries, all three *Cache* operator placement strategies presented in Section 3 produce the same plan: *Cache-Ship-Scan*. Each method involves reading the value of one attribute of the object to which the method is bound. The size of the primary and secondary cache are chosen such that all relevant objects fit in both. We ran each application program ten times (beginning with an empty cache each time) and report on the average running times.

	UDB	Notes	WWW
no caching	47.8	22.9	3538.5
traditional caching	22.9	18.2	1762.3
enhanced caching	2.2	12.7	11.9

Table 3: Total Running Time [secs]

Table 3 shows the results. As expected, *enhanced caching* wins in all cases. The gains are particularly pronounced for the WWW application because interaction with the WWW database, as required to fault in objects, is particularly expensive—even if the WWW server is only lightly loaded and has all information available in main memory. The savings in cost are relatively low for the Notes application because faulting in objects from the Notes database is quite cheap so that the cost of query processing dominates the overall cost of the application in this case. In all cases, *traditional caching*, which faults in objects when they are used for the first time as part of a method invocation, beats *no caching* because it saves the cost of interacting with the data sources for the second method invocation.

In this experiment, the application program accesses *all* objects returned by the query; i.e., $F = 1$. For smaller F, the savings obtained by traditional and enhanced caching are less pronounced. As mentioned in Section 3.3.2, the benefit increases linearly with F; in the extreme case, for $F = 0$, no caching and traditional caching have the same running time as enhanced caching (in fact, a little better).

5.3 Experiment 2: Query Planning Times

The next experiment studied the planning times of the three *Cache* operator placement strategies. The two parameters that impact the planning time most are the number of collections involved in the query and the number of candidate collections. Our queries join collections stored in UDB and Notes. We varied the number of collections involved in the query and in all cases, all collections were considered candidate collections. Thus, these queries can be seen as tough cases which are expensive to optimize.

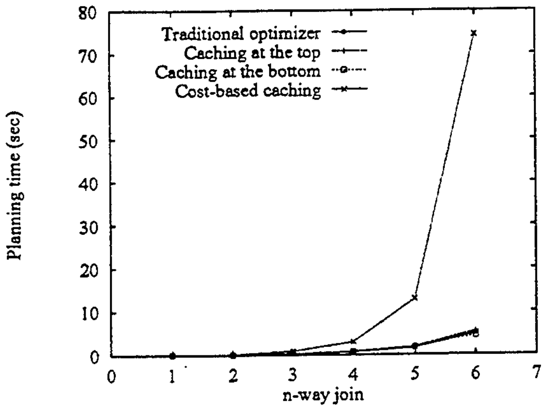


Figure 5: Planning Times for UDB/Notes Queries

Figure 5 shows the resulting planning times for each of the three approaches presented in Section 3. As a baseline, we also show the running time of a traditional optimizer that does not generate plans with *Cache* operators. Again, there are no surprises. The full-fledged cost-based approach becomes prohibitively expensive if there are more than four candidate collections in a query. At this point one of the two heuristics or the variants proposed in Section 3.4 should be used. Up to that point, however, the cost-based approach has negligible overhead and can safely be used. Comparing the "caching at the bottom," "caching at the top," and "traditional optimizer" lines, we see that the two heuristics have virtually no overhead.

5.4 Experiment 3: The Right Caching Decisions

The last set of experiments demonstrates the need to carry out cost-based *Cache* operator placement in certain situations. The experiments show: 1) how a *Cache* operator at the top can increase the cost of the other operators that sit below; 2) the overhead introduced by unnecessarily caching a large number of objects when a *Cache* operator is placed at the bottom; 3) the need to avoid flooding the secondary cache with irrelevant objects; and 4) that it is not always beneficial to have *Cache* operators for all candidate collections, even when accessing slow sources. We used queries over collections from the UDB and WWW databases. The queries and the best execution plan for each query are presented in Figure 6. "Caching at the top" works best for the first query; for the second query, "caching at the bottom" works best; and for the third query, no *Cache* operator at all should be generated. We again measured the total execution time of three simple application programs that each execute one of these queries and invoke one method on each object returned by that query. The method simply reads the value of one attribute. The size of the primary cache was set to 1000 objects which is more than enough to hold all objects involved during method invocations. For the first query (Q1), we studied two configurations for the secondary cache: (a) *medium*, with a capacity of 1000 objects, and (b) *large*, with a capacity of 6000 objects. We varied the size of the secondary cache for Q1 in order to study the implications of loading the cache with irrelevant objects, in particular for the "caching at the bottom" approach. For the other two queries, a *medium* secondary cache was sufficient in all cases, so we only show the results obtained using such a *medium* secondary cache.

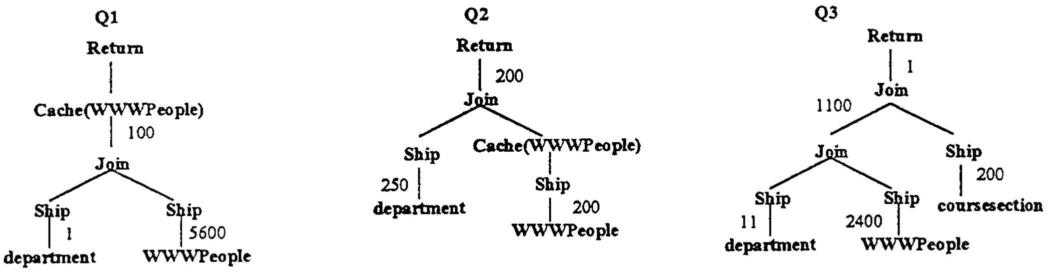
	Q1(large)	Q1(med)	Q2	Q3
no caching	405.5	405.5	842.5	129.2
traditional caching	405.5	405.5	842.7	129.9
caching at the top	71.3	71.3	49.8	177.5
caching at the bottom	76.0	415.8	34.9	141.9
cost-based caching	71.4	71.4	35.1	130.7

Table 4: Total Running Time [secs]
size of sec. cache: medium=1000 obj.; large=6000 obj.

Table 4 shows the results. We can see that the cost-based approach to loading the cache with query results shows the overall best performance, making the right caching decisions in all situations. The "caching at the top" approach, as expected, makes suboptimal decisions for Q2 and Q3, and the "caching at the bottom" approach makes suboptimal decisions for Q1 and Q3. The "caching at the bottom approach" shows particularly poor performance if it floods the secondary cache, so that few relevant objects are loaded as a by-product of executing the query (Q1 with a medium-sized secondary cache). "Caching at the bottom" is never much worse than traditional caching or no caching at all, and it can, therefore, be seen as a *conservative* method of extending today's database systems to load a cache with query results. The "caching at the top" heuristic, on the other hand, is as much as 37% more expensive than traditional caching in our experiments, and could easily be more. In these experiments, traditional caching and no caching show approximately the same performance because every result object is accessed exactly once as part of the method invocations.

6 Related Work

Most work on data processing in distributed systems has focused either on query processing or on caching, and most middleware systems today are built in such a way that query processing does not affect caching and vice versa. For example, SAP R/3 [BEG96, KKM98] is a very popular business administration system that supports the execution of (user and pre-defined) queries and methods, processing applications that involve both as described in Section 2.1. Persistence [KJA93] is a middleware system that enables the development of object-oriented (C++, Smalltalk, etc.) applications on top of a relational database system. That system typically pushes down the execution of queries to the relational database system and executes methods in the middleware using caching. Query processing and caching do not interact in



```
select p.oid
from WWWPeople p, department d
where p.WWWdno=d.dno and
        p.WWWcategory='staff' and
        p.WWWname like 'staffName1%' and
        d.name = 'deptname59'
```

```
select p.oid
from WWWPeople p, department d
where p.WWWdno=d.dno and
        p.WWWcategory='professor' and
        p.WWWdepartment like 'deptname25%'
       and d.dno<251
```

```
select p.oid
from WWWPeople p, department d, coursesection s
where p.WWWdno=d.dno and
        p.WWWid=s.instructorid and
        p.WWWcategory='professor' and
        p.WWWname like 'professorName2%' and
        d.name like 'deptname11%' and d.dno<120 and
        s.deptno=19
```

Figure 6: Benchmark Queries for Experiment 3

either system, so both would benefit from the techniques presented in this paper.

Database systems that have a *data shipping* architecture naturally load a cache with query results; examples are most object-oriented database systems such as O_2 [D+90]. These systems bring all the base data to the middleware (or client) to evaluate a query and that base data is then cached for subsequent queries and methods, if the cache is large enough. In some sense, data shipping, therefore, corresponds to the "caching at the bottom" approach – however, there is no *Cache* operator pull-up and no way to execute joins at data source(s). This causes data shipping to perform poorly for many types of queries [FJK96].

Another experimental database system that supports query processing and caching is KRISYS. In an early version which was targeted for engineering applications, KRYSIS used queries to load the cache with relevant objects [HMNR95], as proposed in our work. However, that version only supported a variant of the "caching at the top" approach (without *Cache* operator pushdown). In a more recent version [DHM+98], KRYSIS supports predicate-based caching. Predicate-based caching [KB94], like view caching [Rou91] and semantic caching [DFJ+96], makes it possible to cache the results of queries. The purpose of predicate-based caching, however, is to use the cache in order to answer future queries (rather than for methods). Hence, it requires significantly more complex mechanisms for tracking cache contents, and is not geared for the lookup of individual objects.

Two further lines of work are relevant. The first is cache investment [FK97]. Cache investment also extends a query processor to make it cache-aware. Again, however, the purpose of cache investment is to load the cache of the middleware in such a way that future queries (rather than methods) can be executed efficiently. The second related line of work is prefetching [PZ91, CKV93, GK94]. The purpose of prefetching is to bring objects into the cache before they are actually accessed. Prefetching, however, is carried out as a separate process, independent of query processing.

7 Conclusion

In this paper, we showed that caching objects during query execution dramatically speeds up applications that involve both queries and methods in a middleware (or client server) environment. The performance wins that can be achieved by this method are huge; they are particularly high in environments in which interactions with the data sources are very expensive; e.g., data sources on the Internet. In certain scenarios, loading a cache with query results in this way is even necessary; such a situation arises in heterogeneous database environments in which some data sources are not able to respond to requests for individual objects.

To implement our approach we extended the cache manager and the query processor of a middleware system. We used a double caching scheme to reduce the overhead of our approach and to avoid flooding the primary cache with (useless) objects as a by-product of query execution. We explored three alternative ways of extending the query processor: "caching at the top," "caching at the bottom," and "cost-based caching." The first two approaches are simple heuristics which can be easily incorporated in an existing query processor and which typically do not increase query optimization times; however, the "caching at the top" approach can result in substantially increased query execution times, while the "caching at the bottom" approach may cache many useless objects, thereby causing additional overhead and providing no benefit if the cache is too small. The third approach is significantly more complex to implement and increases optimization times of complex queries substantially, but is always able to make the best decisions of the three approaches. Based on these observations, we propose to use the full "cost-based" approach for simple queries that involve no more than four collections and heuristics for more complex queries. In the future, we plan to investigate the tradeoffs of optimization time and application performance for some of the variants described in Section 3.4.

8 Acknowledgements

We thank Mary Tork Roth, Peter Schwarz and Bart Niswonger for their help with this work.

References

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compiler Construction*, volume II. Addison-Wesley, 1989.

[BEG96] R. Buck-Emden and J. Galimow. *SAP R/3 System, A Client/Server Technology*. Addison-Wesley, Reading, MA, USA, 1996.

[CDN+97] M. Carey, D. DeWitt, J. Naughton, M. Asgarian, J. Gehrke, and D. Shah. The bucky object-relational benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–146, Tucson, AZ, USA, May 1997.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CKV93] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 43–53, Washington, DC, USA, May 1993.

[D+90] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.

[DHM+98] S. Deßloch, T. Härder, N. Mattos, B. Mitschang, and J. Thomas. KRISYS: Modeling concepts, implementation techniques, and client/server issues. *The VLDB Journal*, 7(2):79–95, April 1998.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 149–160, Montreal, Canada, June 1996.

[FK97] M. Franklin and D. Kossmann. Cache investment strategies. Technical Report CS-TR-3803, University of Maryland, College Park, MD 20742, May 1997.

[GK94] C. A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, pages 351–364, Cambridge, United Kingdom, March 1994. Springer-Verlag.

[HKU99] L. Haas, D. Kossmann, and I. Ursu. An investigation into loading a cache with query results. Technical report, IBM Almaden, San Jose, CA, March 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HMNR95] T. Härder, B. Mitschang, U. Nink, and N. Ritter. Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen. *Informatik – Forschung und Entwicklung*, 10(2):55–72, May 1995.

[KB94] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, pages 229–238, Austin, TX, USA, September 1994.

[KJA93] A. Keller, R. Jensen, and S. Agrawal. Persistence software: Bridging object-oriented programming and relational databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 523–528, Washington, DC, USA, May 1993.

[KK95] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *The VLDB Journal*, 4(3):519–566, August 1995.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[Mos92] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Software Eng.*, 18(8):657–673, August 1992.

[PZ91] M. Palmer and S. Zdonik. FIDO: A cache that learns to fetch. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 255–264, Barcelona, September 1991.

[ROH98] M. Tork Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, GB, September 1998.

[Rou91] N. Roussopoulos. The incremental access method of view cache: Concepts, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, September 1991.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.